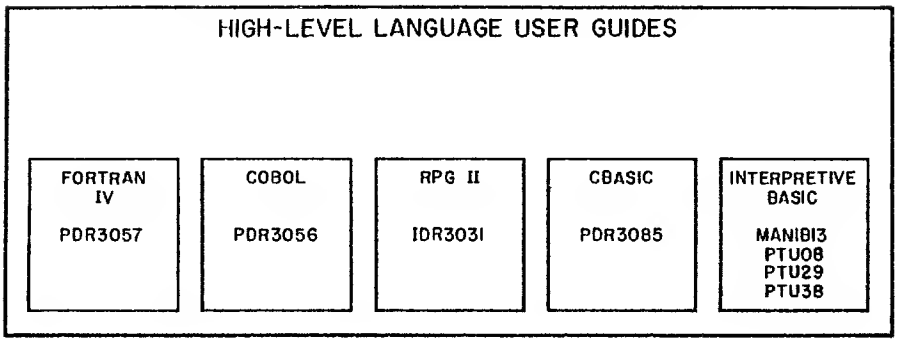
PRIME

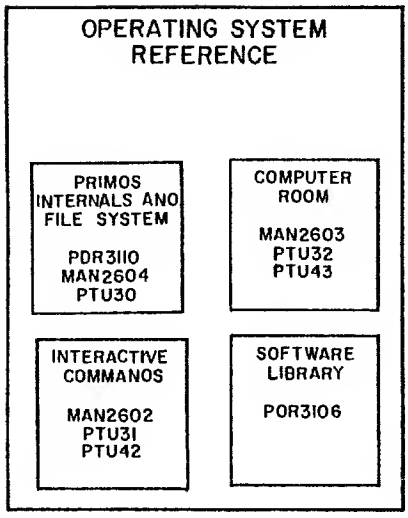
Preliminary Documentation Release

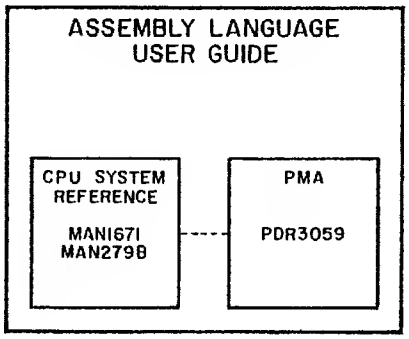
PDR3061 REFERENCE GUIDE, MULTIPLE INDEX DATA ACCESS SYSTEM (MIDAS)

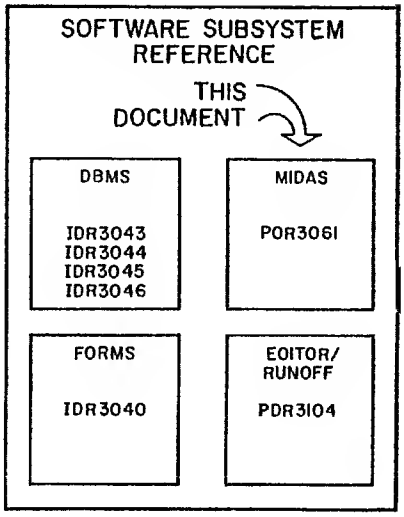
P/N 3064-001

PRIME SOFTWARE DOCUMENTATION FAMILY









This is the reference guide for MIDAS -- Prime's Multiple Index Data Access System. It contains the detailed information needed by a FORTRAN, COBOL or RPG programmer in order to:

- Build a MIDAS template file with the CREATK interactive utility.
- Build MIDAS data files, either with the KBUILD interactive utility or with the BILD$R, PRIBLD and SECBLD user-program subroutines.
- Maintain a MIDAS data base using the FORTRAN-compatible MIDAS data access subroutines (FIND$, etc.). (COBOL and RPG users use the protocols for these languages.)
- Restructure an existing application file with the interactive utilities REMAKE, REPAIR and KIDDEL.
- Modify MIDAS parameter files and COMMON blocks to meet special user needs.

The guide concludes with a detailed analysis of the planning,, implementation and use of MIDAS in an order entry and back-order control application.

All correspondence on suggested changes to this document should be directed to:

Max Goudy, Technical Writer
Technical Publications Department
Prime Computer, Inc.
145 Pennsylvania Avenue
Framingham, Massachusetts 01701

Acknowledgements

We wish to thank the members of the REFERENCE GUIDE, MULTIPLE INDEXED DATA ACCESS SYSTEM team and also the non-team members, both customer and Prime, who contributed to and revised this PDR.

PRIME DOCUMENTATION TYPES

IDR Initial Documentation Release: provides usable, accurate advanced information without regard to style and format.

PDR Preliminary Documentation Release: provides more complete and accurate information about the product, but is not in final format.

FDR Final Documentation Release: a complete product description: edited, formatted and produced at a high standard of graphic quality.

MAN Manual: early reference documents to be phased out by PDR's and FDR's.

PTU Prime Technical Update: interim updates to existing documents.

First Printing November 1977

CONTENTS

CONTENTS (Cont'd)

CONTENTS (Cont'd)

SECTION 1

INTRODUCTION TO MIDAS

PURPOSE

MIDAS (Multiple Index Direct Access System) provides a series of programs and subroutines for the creation and maintenance of Keyed-Index Direct Access (KI/DA) files.

Keyed-Index files are sometimes referred to as ISAM (Indexed Sequential Access Method) files. MIDAS gives the user the advantages of ISAM files plus additional useful features, all of which are described in this document. Figure 1-1 gives a functional overview of MIDAS. This schematic attempts to relate the tasks of creating a file template, building a data file, maintaining a file, and the data access.

MIDAS usage falls into four areas (see Figure 1-1)

- Creating/modifying the template - the user defines the data file, indices, etc. (CREATK)
- Building the data file - data existing in a text or binary file are converted to a MIDAS file. (KBUILD)
- Maintaining the file - data entries are added, deleted, changed, or viewed.
- Performing housekeeping - files are restructured after significant maintenance (REMAKE), deleted in part or full (KIDDEL), or rebuilt after crashes (REPAIR).

MIDAS AND THE FILE SYSTEM

At Rev. 14, MIDAS has been converted to the new file system. With the change the format of MIDAS's internal pointers has also changed. The change has been implemented so that accessing a MIDAS file with the Rev. 14 (or later) version of MIDAS automatically converts the file to new pointers. NOTE, however, that once a file has been accessed by a new version of MIDAS, it can no longer be accessed by older versions.

USING MIDAS

Generally, the first task a user is concerned with is the creation of a template (file descriptor) for the MIDAS file. A program, CREATK, is provided for that purpose. (Refer to Section 2 for details of CREATK usage.) CREATK, along with REMAKE, can also be used to modify and update the template. MIDAS also provides a program, KBUILD, and, alternatively, provides a series of subroutines; PRIBLD, SECBLD, and BILD$R, to build a MIDAS data file from an input user-data file. (refer to Sections 7 and 8 for further information on data file building.) The programs REMAKE, REPAIR, and KIDDEL provide a means for ensuring MIDAS file integrity. For further information, refer to Sections 4,5, and 6. Finally, a series of subroutines are available for access, maintenance, and retrieval of the data records in a MIDAS file. They are: ADD1$, DELET$, FIND$, LOCK$, NEXT$, and UPDAT$. (Refer to Section 9 for further details.)

Creating and Modifying the Template

The interactive program CREATK allows the user to build, examine, and modify the MIDAS file template. This template contains the information the MIDAS programs and subroutines required to build and maintain the data file and its associated index file(s) and directories. When constructing the template, the user specifies filename, direct access support (if supplied), block, length, and index requirements (both primary index and secondary indices, if any). For many parameters, the system will supply default values in lieu of the user's specifications if so desired.

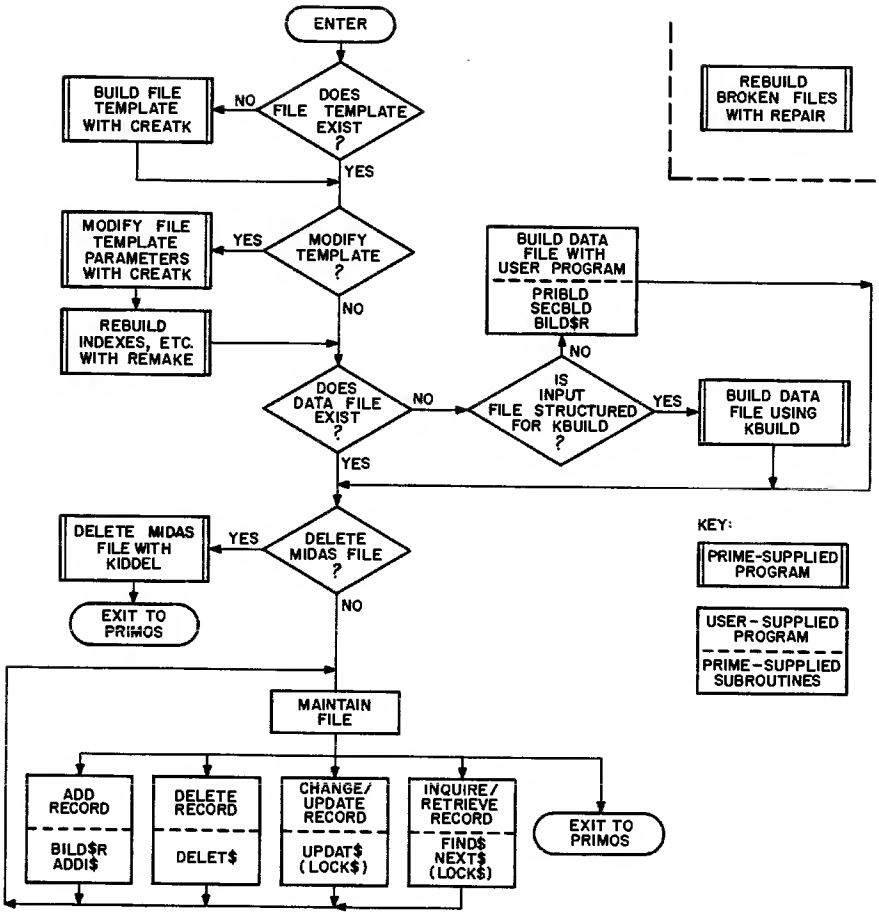


Figure 1-1. Functional Overview of MIDAS

Example CREATK Dialog

The following CREATK dialog shows the creation of the template for a new file. User response is underlined.

```
OK, CREATK
GO
MINIMUM OPTIONS? YES

FILE NAME? POLITIC
NEW FILE? YES
DIRECT ACCESS? NO

DATA SUBFILE QUESTIONS

KEY TYPE: A
KEY SIZE = : W 2
DATA SIZE = : 40

SECONDARY INDEX

INDEX NO.? 1

DUPLICATE KEYS PERMITTED? YES
KEY TYPE: A
KEY SIZE = : W 1
USER DATA SIZE = : 2

INDEX NO. 2

DUPLICATE KEYS PERMITTED? YES
KEY TYPE: A
KEY SIZE = : W 2
USER DATA SIZE = : 40

INDEX NO.? CR

OK,
```

Building the Data File (Single-User Environment)

The MIDAS data file may be constructed with the Prime-supplied program KBUILD (Refer to Section 3) or the user may create a program with the PRIME-supplied subroutines BILD$R, PRIBLD, SECBLD (Refer to Section 7). Using KBUILD is simpler but it has certain restrictions; for example, input data files and the resulting output MIDAS data file must have fixed length records.

Maintaining and Using the MIDAS File (Multi-User Environment)

A number of subroutines are supplied to enable the programmer to make effective use of the MIDAS file. These subroutines are designed to allow more than one user to access the data file simultaneously. The lockout subroutine protects data entries from attempts at simultaneous changes/deletions. All the subroutines require the file PARM.K in the UFD name SYSCOM to be inserted in the user program by the $INSERT command (Refer to Section 8). A summary of these MIDAS subroutines follow.

Subroutine Functions

ADD1$	adds a data entry to the file and modifies the index files appropriately. Insertion is by primary key only; the file is locked during insertion.
DELET$	deletes a data entry and modifies the index file(s) accordingly. Deletion may not occur if the data entry is locked.
FIND$	locates a data entry and reads its contents into a buffer. Look-up is by primary and secondary key(s). If data entries exist with the same secondary key (synonyms), the oldest data entry (i.e., first one in the file) is retrieved.
NEXT$	retrieves the data entry with the next higher key. Search may be on a primary or secondary key. This subroutine allows synonyms which are not oldest to be accessed.
LOCK$	locates a data entry and locks it, if not already locked. The data entry is only unlocked by a successful call to UPDAT$.
UPDAT$	rewrites a data entry. This subroutine should not be called before a successful call to LOCK$.

GLOSSARY OF TERMS

The following paragraphs describe a number of terms and conventions used throughout this manual. They are arranged in sequence to help the reader progress from fundemental to more complex concepts.

MIDAS File

A KI/DA (Keyed Index-Direct Access) File.

KI/DA File

A collection of subfiles under a segment directory. In particular:

1. A file descriptor subfile (Segment Ø)
2. One or more index subfiles (Segments 1-184)
3. A data subfile (Segments 185)

Segment Directory

A file whose entries are pointers to other files. The other files are called segments or segment subfiles.

For information on files and file types refer to the File System User Guide (MAN 2604) and the File Management System Reference Guide (PDR3110).

Segment

A Segment (or "segment subfile") is a SAM or DAM file that can be accessed only through its associated segment directory. Segments are numbered 0, 1, 2, . . . etc. Most of MIDAS' segments are DAM files.

Index Subfile

Physically, an index subfile consists of one or more segments. The first segment is SAM and is required. The other segments are DAM files and are created only as needed. Logically, an index subfile consists of an index descriptor block and one or more index blocks. The resulting index is a tree structure. The index descriptor block and one index block are contained in the required first segment of the index. All other index blocks are contained in the other segments of the index.

Figure 1-2 shows the layout of the index subfiles within a MIDAS file structure.

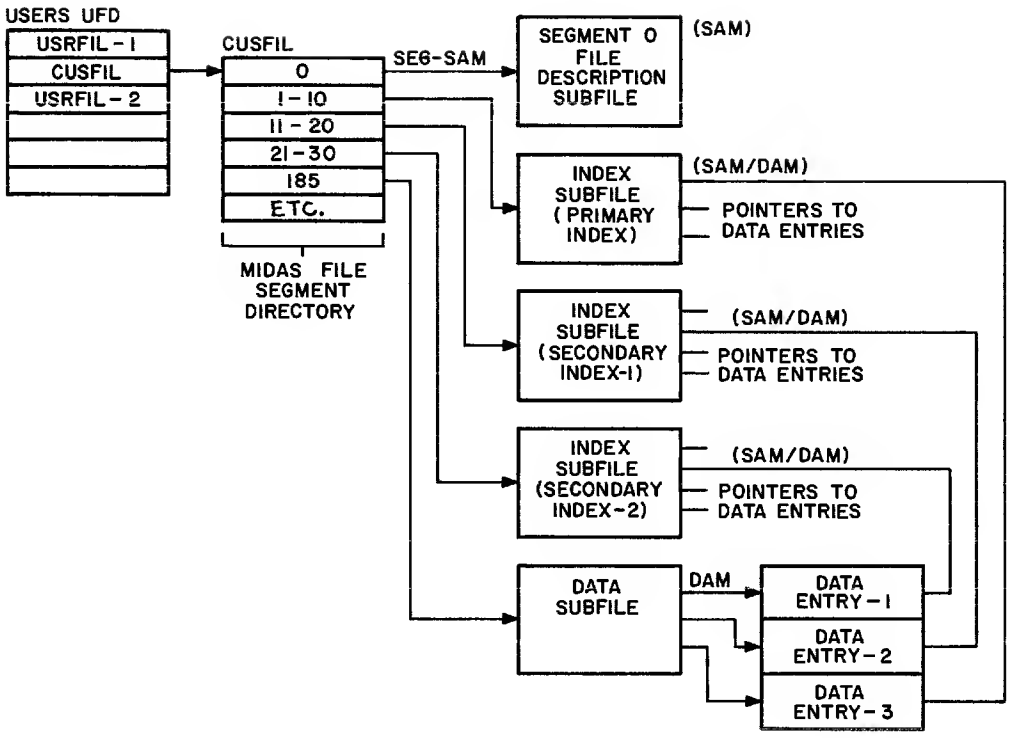


Figure 1-2. Typical MIDAS File Structure

Index Descriptor Block

A local guide to the index subfile. It is used when adding an entry to the index or deleting an entry from it.

Index Block

A collection of "index entries", together with a search rule for searching the entries in the block. The search rule is contained in Control words at the beginning of the block. The index entry is used to determine what to search next.

Index Entry

A key (key value) and a three word pointer to either another index block or a data entry. For secondary indexes only, it may also contain optional secondary data.

Key, Key Value, Key Field

Key, Key value, Key field refer to a value that is used to look up something in an index. For example, in an employee file for PRIME computer 'MAX GOUDY' and 'ROGER THORT' might be keys. Then somewhere, in the appropriate index would be an "index entry containing "MAX GOUDY, ptr" and "ROGER THORT, ptr".

Sometimes "key" is modified by reference to an index number. For example:

"Primary key" a key used with a primary index (Index 1)
"Secondary key" a key used with a secondary index (Index 1)

Secondary Data

Information separate from the data record that is stored in a secondary index entry. Secondary data may be retrieved only through the secondary index entry.

Template File

A Template (KI/DA) file (See Figure 1 - 2) is a segment directory plus

1. A file descriptor subfile (Segment Ø)
2. A Primary index subfile which contains only an index descriptor block and a "seed" index block.
3. Optional secondary index subfiles each containing only an index descriptor block and a "seed" index block.

Overflow

Eventually, so many overflow entries may be accumulated that file access slows down considerably. In this case the REMAKE utility can be run to restructure the file and incorporate all overflow entries. Overflow is a collection of sorted chains of index entries stored in index blocks that are effectively merged into the main body of the index when the index is traversed sequentially. Overflow entries accumulate during access and use of a MIDAS file.

Space Recovery

Sometimes a user will have deleted many records from the index and will wish to retrieve the lost space. The REMAKE utility can be run to compress the file and recover all unused records.

SECTION 2

CREATK UTILITY

CREATK is a interactive utility module that creates a template for a MIDAS keyed-index file. CREATK may also be used to modify the template of an existing file, or obtain information about the structure of a file. The functions available are:

- Create a new MIDAS file template.
- Modify index or data descriptions for an existing file.
- Add new secondary index subfiles to an existing file.
- Display existing index or data descriptions.

CREATK FUNCTIONS

CREATK generates the minimum requirements for a MIDAS file: a segment directory, a file descriptor subfile, and one-level primary index subfile that contains the index descriptor block and an empty last level index block. Additionally, if the file is organized for direct access, the data segments are allocated and initialized. Similarly, for each secondary index that is defined, an index subfile (i.e., an index descriptor block and empty last level index block) are created.

CREATK can define additional secondary index subfiles once the basic file has been defined, or can change the description of existing primary or secondary index subfiles. If the description of an existing index subfile is modified, this is reflected only in the description in the file descriptor subfile. The existing index is not affected and may continue in use until it is convenient to use the REMAKE program to update the index subfile. It is also possible to change the length of index subfile by changing the segment length which can have the effect of making a MIDAS index subfile hold more entries.

CREATK may be used to determine the potential size of a MIDAS file based on the maximum number of records to be stored in the file. Finally, CREATK is used to examine the existing file to determine the number of entries currently in each index or to display the parameters used to create the file, such as key length, block length, data length and so on.

CREATK does not allow the user to destroy a MIDAS file that already exists. If the operating system is properly configured, the file may be examined and modified when the file is being used by more than one user.

Minimum Options

For most MIDAS files, the MINIMUM OPTIONS alternative provides a nearly optimum file design. Users that mix storage module disks with other disk types may need to change the block size for some files, from the default size to a size that optimized access times.

If MINIMUM OPTIONS is selected, all keys have the same length as the full key for the last level index. All index subfiles have their default length specified in the $INSERT file named KPARAM (440 words, as delivered prior to Revision 14; 1024 words thereafter).

Alternative Versions of CREATK

The standard version of CREATK, as delivered, is built to support direct access but not long indexes. The user may build alternate versions to enable or suppress these features. For details, see SELECTING A VERSION OF CREATK at the end of this section.

CREATK DIALOG

Following is a step-by-step description of the basic CREATK dialog. To assist the user in following the dialog, line numbers have been assigned to each step. For every step requiring user response, all the possible answers are shown. A "goes to" statement in parentheses, following each response, indicates where the dialog resumes.

Note

All user input to CREATK must be in UPPER-CASE letters only.

The CREATK dialog is also illustrated in flow-chart form in Figures 2-1 through 2-5.

Invoking CREATK

To commence the CREATK program, the user types:

CREATK

at the terminal. CREATK responds with the first question of the dialog:

MINIMUM OPTIONS?

By replying YES to this question a shortened version of the CREATK dialog is utilized. A NO response enters the expanded version of CREATK. The expanded version of CREATK is used when its optimizing capabilities are required. (For full details see Appendix B.) This section deals only with the MINIMUM OPTIONS path.

File Identification - Lines 0-4 (Figure 2-1).

Reviewing for a moment, a user desiring the MINIMUM OPTIONS CREATK dialog would begin his terminal session as follows:

(Line 0)	OK, CREATK	(Goes to Line 1)
(Line 1)	MINIMUM OPTIONS? YES	(Goes to Line 2)

If the response to line 1 is NO, the CREATK dialog resumes on line 38, which begins the expanded version. Any response other than either YES or NO is invalid and the dialog repeats line 1.

Assuming a YES response to MINIMUM OPTIONS, the CREATK dialog resumes on line 2 as follows:

(Line 2)	FILE NAME	(treename)	(Goes to Line 3)
		(other)	(Repeats Line 2)

The parameters enclosed in the brackets can either describe a pathname or filename with options.

(Line 3)	NEW FILE?	YES	(Goes to Line 4)
		NO	(Goes to Line 23)
		(other)	(Repeats Line 3)

If the FILE NAME given in line 2 is a new file, the response to NEW FILE is YES. If not, the user responds with NO and CREATK skips to line 23. Assuming the response is YES, the dialog resumes on line 4.

Line 4 is optional and is skipped if this version of CREATK was built without Direct Access support. If this is the case, the dialog would resume on line 5. Assuming that Direct Access is supported by CREATK:

(Line 4)	DIRECT ACCESS?	YES	(Goes to Line 17)
		NO	(Goes to Line 5)
		(other)	(Repeats Line 4)

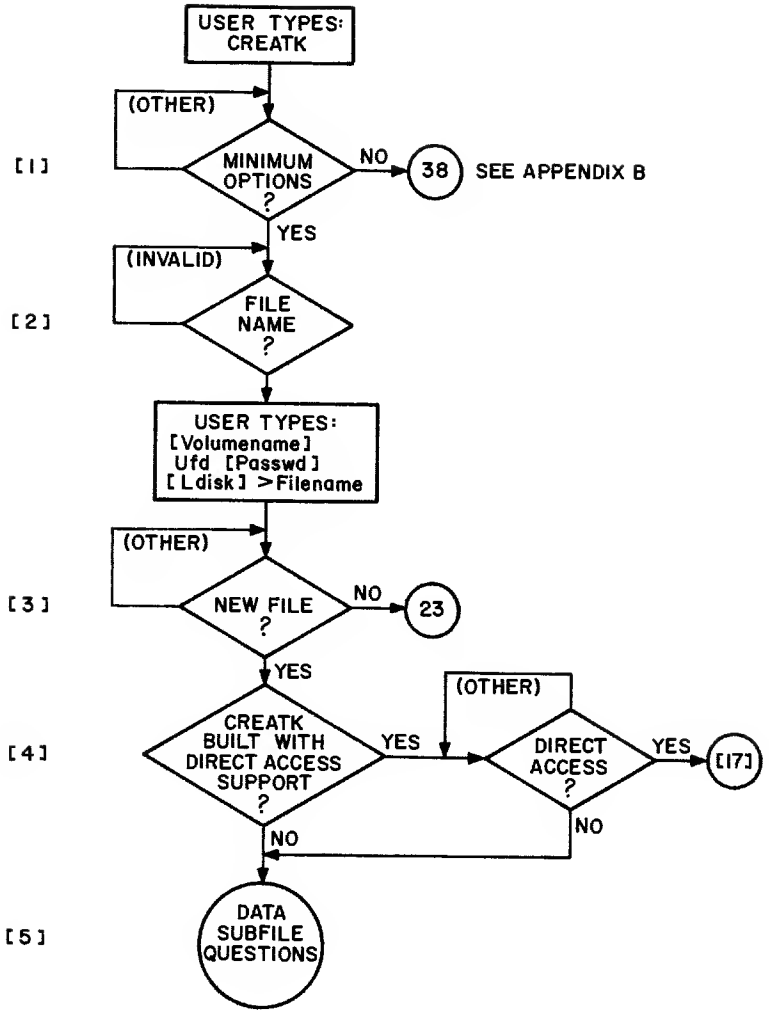


Figure 2-1. CREATK Dialog - File Identification

Data Subfile Questions - Lines 5-9 (Figure 2-2)

(Line 5) DATA SUBFILE QUESTIONS (Goes to Line 6)

No user reply is necessary in line 5. If the response to DIRECT ACCESS is NO, CREATK defines the parameters for data subfiles in lines 6 through 9. Parameters for Direct Access data subfiles are defined in lines 17 through 22. Assuming that the file named by FILE NAME is not for Direct Access:

(Line 6) KEY TYPE :	B	(Goes to Line 7)
	A	(Goes to Line 7)
	I	(Goes to Line 8)
	L	(Goes to Line 8)
	D	(Goes to Line 8)
	S	(Goes to Line 8)
	(other)	(Repeats Line 6)

There are six possible valid responses to line 6. Each indicates a different key type. These key types are as follows:

Response in Line 6 is:	Key Type is:
A	ASCII character string; the key length in bytes or words must be supplied in line 7.
B	Bit string; the key length in bits or words must be supplied in line 7.
I	Signed Short Integer
L	Signed Long Integer
S	Single Precision Floating Point
D	Double Precision Floating Point

Key types A and B require that key length be specified, as they have no predetermined length. Key length is defined to CREATK in line 7. Numeric key types I, L, S and D do not require such specification as their respective lengths are known by definition. If the response to line 6 is either A or B the dialog resumes on line 7 as follows:

(Line 7) KEY SIZE=	:B	(Number of bits/bytes)	(Goes to Line 8)
	W	(Number of words)	(Goes to Line 8)
	(other)		(Repeats Line 7)

KEY SIZE requires two specifications. The first indicates the unit of measure for the key. Either B (bits or bytes) or W (words) can be used. This specification is followed by a space and then the KEY SIZE. For example to indicate a KEY SIZE of 32 bytes, the user would type the following in line 7:

KEY SIZE= :B 32

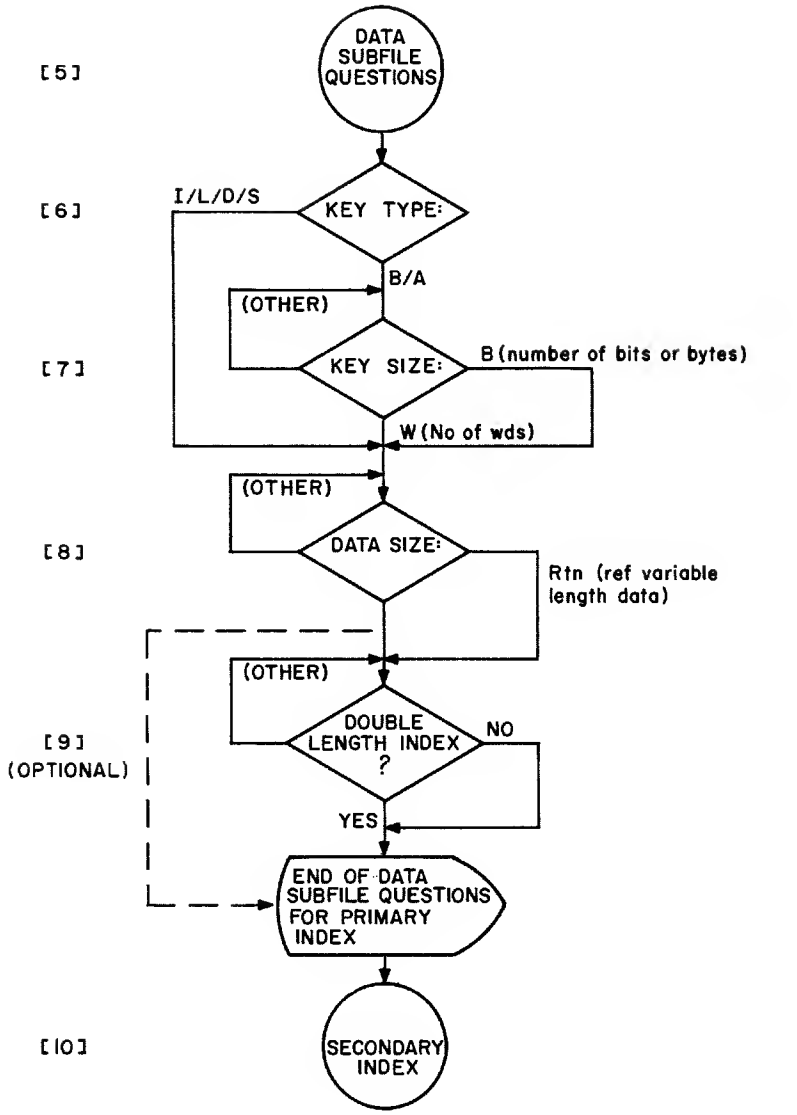


Figure 2-2. CREATK Dialog - Data Subfile Question

The CREATK dialog resumes on line 8 as follows:

(Line 8)	DATA SIZE= :	(Number of words in entry)	(Goes to Line 9)
		(CR)	(Goes to Line 9)
		(other)	(Repeats Line 8)

DATA SIZE refers to the length of the data record in the data subfile. Size is expressed in words and refers to the data entry portion of data record only. It does not include other data subfile fields, such as control words or the primary key value. Typing a CR in line 8 indicates that there are variable length data entries. In either case the dialog resumes on line 9 as follows:

(Line 9)	DOUBLE LENGTH INDEX?	YES	(Goes to Line 10)
		NO	(Goes to Line 10)
		(other)	(Repeats Line 9)

The YES response indicates that a long index is desired. Long indexes are described in detail at the end of this section under SELECTING A VERSION OF CREATK.

Line 9 is optional, and in either case the dialog resumes on line 10.

Secondary Index Questions - Lines 10-16 (Figure 2-3)

(Line 10)	SECONDARY INDEX		(Goes to Line 11)

There is no user input required in line 10.

(Line 11)	INDEX NO.?	(number from 1 to 19)	(Goes to Line 12)
		(CR)	(Goes to Line 22)
		(other)	(Repeats Line 11)

If there are going to be SECONDARY INDEX subfiles, the number of these files to be buile is defined in line 11. Typing a number from 1 to 19 following INDEX NO defines the index number to CREATK. After all the SECONDARY INDEX questions have been answered, CREATK will loop back to line 11 to see whether there are more SECONDARY INDEX subfiles.

Typing a (CR) in line 11 indicates either that there are no SECONDARY INDEX subfiles, or that there are no more. Assuming that there is at least one SECONDARY INDEX, the dialog resumes on line 12 as follows:

(Line 12)	DOUBLE LENGTH INDEX?	YES	(Goes to Line 13)
		NO	(Goes to Line 13)
		(other)	(Repeats Line 12)

Line 12 is optional, depending on the version of CREATK. In either case the dialog resumes on line 13 as follows:

(Line 13)	DUPLICATE KEYS PERMITTED?	YES	(Goes to Line 14)
		NO	(Goes to Line 14)
		(other)	(Repeats Line 13)

(Line 14)	KEY TYPE :	B	(Goes to Line 15)
		A	(Goes to Line 15)
		I	(Goes to Line 16)
		L	(Goes to Line 16)
		D	(Goes to Line 16)
		S	(Goes to Line 16)
		(other)	(Repeats Line 14)

There are six possible valid response to Key TYPE. Refer to the list following line 6 for a full explanation. Key types B and A require a key length to be specified. Key types I, L, D and S do not. The key length, when required, is entered on line 15.

(Line 15)	KEY SIZE = :	B	(Number of bits or bytes)	(Goes to Line 16)
		W	(Number of words)	(Goes to Line 16)
		(other)		(Repeats Line 15)

The KEY SIZE requries two specifications, Refer to the explanation following line 7.

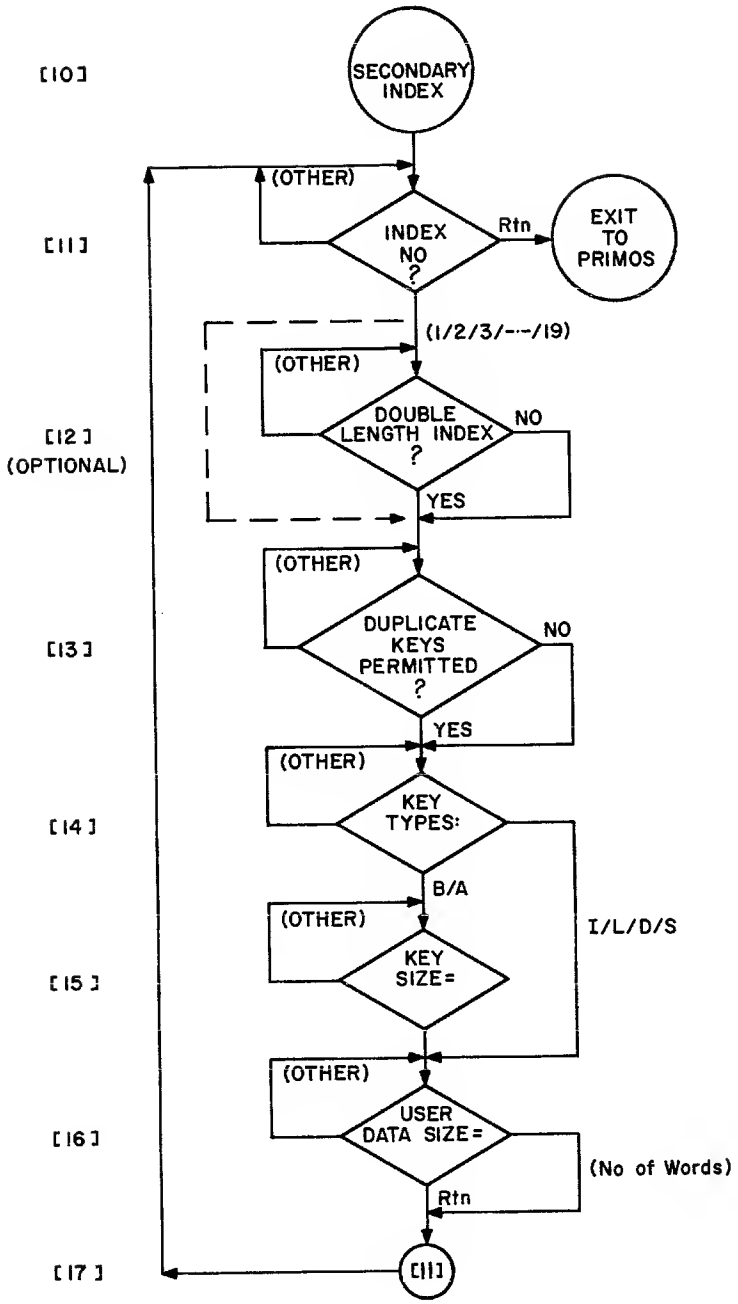


Figure 2-3. CREATK Dialog - Secondary Index Questions

(Line 16)	SECONDARY DATA SIZE=	: (Number of words)	(Loops back to Line 10)
		(CR)	(Loops back to Line 10)
		(other)	(Repeats Line 16)

SECONDARY DATA SIZE refers to the length of the data entry included along with the key. Size is expressed in words. A response of either Ø or (CR) indicates that there is no SECONDARY DATA.

Reviewing for a moment, the preceding 13 lines dealt with files without Direct Access support. If instead on line 4 the response is YES the CREATK dialog would resume at line 17. The following four lines deal specifically with files set up for Direct Access.

Data Subfile Questions - Lines 17-22 (Figure 2-4)

(Line 17) DATA SUBFILE QUESTIONS		(Goes to line 18)

No response is necessary in line 17.

(Line 18) KEY TYPE :	B	(Goes to Line 19)
	A	(Goes to Line 19)
	I	(Goes to Line 20)
	L	(Goes to Line 20)
	D	(Goes to Line 20)
	S	(Goes to Line 20)
	(other)	(Repeats Line 18)

There are six possible valid responses to line 18. Refer to the list following line 6 for a full explanation. Key types B and A require the key length to be specified. Key types I, L, D and S do not. When required the key length is specified in Line 19.

(Line 19) KEY SIZE=	:B	(Number of bits or bytes)	(Goes to Line 20)
	W	(Number of words)	(Goes to Line 20)
	(other)		(Repeats Line 19)

KEY SIZE requires two specifications. Refer to the explanation following line 7.

(Line 20) DATA SIZE=	:	(Number of words in entry)	(Goes to Line 21)
		(CR)	(Goes to Line 21)
		(other)	(Repeats Line 20)

DATA SIZE refers to the length of the data record in the data subfile. Size is expressed in words and refers to the data entry portion of the data record only. It does not include other data subfile fields, such as control words or the primary key value. Typing a CR in line 20 indicates that there are variable length data entries.

(Line 21) NO OF ENTRIES TO ALLOCATE?		(Numeric)	(Goes to Line 22)
		(other)	(Repeats Line 21)

(Line 22) DOUBLE LENGTH INDEX?	YES		(Loops back to Line 10)
	NO		(Loops back to Line 10)
	(other)		(Repeats Line 22)

Line 22 is optional, depending on the version of CREATK. In either case the dialog loops back to line 10 for additional SECONDARY INDEX specifications.

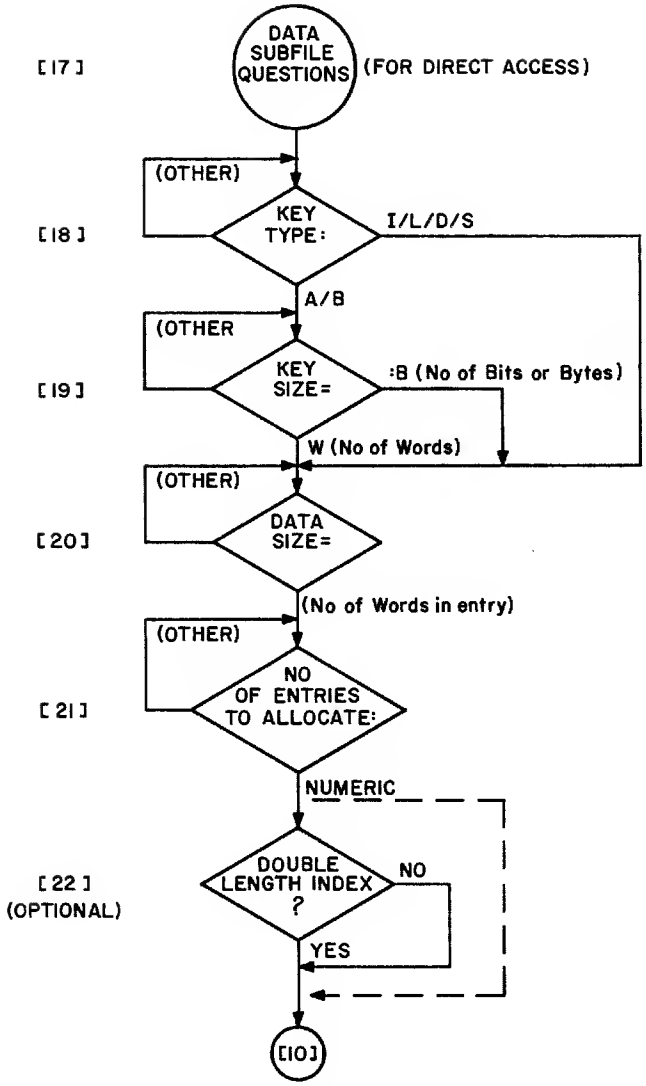


Figure 2-4. CREATK Dialog - Direct Access Questions

Existing File Modification - Lines 23-36 (Figure 2-5)

Reviewing for a moment, lines 4 through 22 dealt with new files only. If the response in line 3 is:

NEW FILE? NO

the CREATK dialog would resume on line 23. The remainder of the MINIMUM OPTIONS dialog deals with operations performed on existing MIDAS files.

(Line 23)	FUNCTION?	MODIFY	(Goes to Line 24)
		ADD	(Goes to Line 26)
		DATA	(Goes to Line 27)
		PRINT	(Goes to Line 28)
		HELP	(Goes to Line 30)
		FILE	(Goes to Line 31)
		QUIT	(Returns to PRIMOS)
		USAGE	(Goes to Line 32)
		LUSAGE	(Goes to Line 33)
		VERSION	(Goes to Line 34)
		EXTEND	(Goes to Line 35)
		SIZE	(Goes to Line 36)
		(other)	(Repeats Line 23)

There are twelve valid responses to FUNCTION; each is described below.

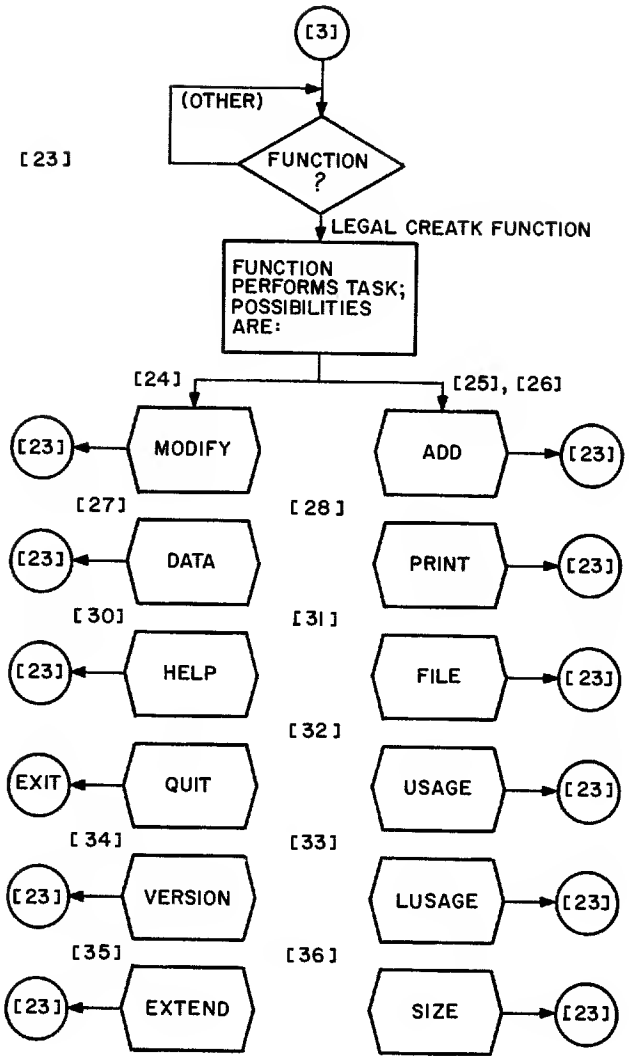


Figure 2-5. CREATK Dialog Conclusion

QUIT Function

QUIT terminates CREATK and returns the user to PRIMOS command level. A (CR) or a blank in response to a dialog prompt message has the same effect.

QUIT is executed by typing QUIT to the FUNCTION question on line 23. Once QUIT has been executed, reentry into CREATK can only be accomplished by a full restart.

(Line 24)	INDEX NO.?	(A number from 1-19)	(Goes to Line 25)
		(CR)	(Returns to PRIMOS)
		(other)	(Repeats Line 24)

MODIFY Function

MODIFY enables a user to change specific parameters and/or data from a specified INDEX. Modifications supported include:

1. Change block length
2. Add data
3. Remove data
4. Change key lengths (All index levels except last)
5. Change support of duplicate key occurrences.
6. Change index to long index (if supported by the CREATK version in use).

(Line 25)	The remainder of the MODIFY dialog is similar to adding either a primary or secondary index subfile except KEY SIZE may not be modified. If no subfile exists to be modified, an error message is displayed.	(Returns to Line 23)

ADD Function

(Line 26)	The ADD dialog adds a secondary index to an already existing file. The ensuing dialog is similar to creating a secondary index (Lines 10 through 16). If the secondary index already exists, an error message is displayed.	(Returns to Line 23)

DATA Function

(Line 27) Data may be redefined with the DATA function. The length of a data entry may be made shorter or longer. The DATA dialog is similar to the data subfile questions (Line 17). (Returns to Line 23)

PRINT Function

(Line 28) INDEX NO.?

(Number from 1 to 19)	(Goes to Line 29)
(DATA)	(Goes to Line 29)
(other)	(Repeats Line 28)

(Line 29) The current configuration of the specified subfile is displayed. The configuration is taken from the file descriptor subfile. (Returns to Line 23)

The PRINT function enables the user to obtain various descriptions of the index structures. For index subfiles, the following information is provided by PRINT:

1. Number of segments allocated
2. Index capacity (entries which can be indexed - Rev 14)
3. Key type
4. Number of index levels (as of last REMAKE)
5. For each level of indexing:
   - Block size
   - Key length
   - Number of control words
   - Maximum number of entries per block
   - Length of an index entry
   - Number of blocks in the level (as of last REMAKE)

For data subfiles, the following information is provided by PRINT:

1. File type (Keyed Index or Direct Access)
2. Number of index subfiles defined
3. Number of entries indexed (as of last REMAKE)
4. Entry size
5. Key size

HELP Function

(Line 30) HELP prints a summary of CREATK functions at the users terminal. (Returns to Line 23)

FILE Function

(Line 31) The FILE function allows specification of a new working file without leaving and reentering CREATK. FILE specifies that the old file is to be closed and a new session may begin in the new file. CREATK resumes the dialog at the NEW FILE question on line 2. (Returns to Line 2)

USAGE Function

(Line 32) The USAGE function allows the user to display the number of records currently available through any defined index. The number of records are displayed as; ENTRIES INDEXED, ENTRIES IN OVERFLOW and ENTRIES DELETED. These values are summed to provide the TOTAL ENTRIES IN FILE. Additionally, the number of blocks in overflow and the length of the longest overflow chain are displayed if these are nonzero. (Returns to Line 23)

LUSAGE Function

(Line 33) The LUSAGE function provides the same information as USAGE. It also prints additional information concerning the version of MIDAS last used to modify the file (Revision 12 or later). (Returns to Line 23)

VERSION Function

(Line 34) The VERSION function displays the version of MIDAS used to create the file, and the version of CREATK used to create the file and the parameters used. (Returns to Line 23)

EXTEND Function

(Line 35) The EXTEND function allows the user to change the length of the segment and/or segment directory. The user is asked the length of the segment directory in words and the length of each segment in words. If either Ø or (CR) are supplied, the default values in effect for CREATK are used.

The length of a segment directory for a MIDAS file has previously been set to 440 segments. As of revision 14 this has been changed to 512 segments. The main reason for changing the length of the segment directory is to increase the size of the data subfile. The data subfile begins in segment 185, hence the size of the data subfile is (segdir length - 185) *segment length. There is no need to increase the length of the segment directory if there is sufficient data subfile space.

The length of a segment is expressed in words. Prior to revision 14, segments held 193600 words. As of revision 14 the default length is 524288 words. The shorter length assumed 440 word disk records, the longer 1024 word disk records. This length may be changed so that a user may increase the capacity of a file or optimize a file for the type of disk containing it.

The main reason for changing the length of a segment is to increase the size of an index. This change is then implemented across all segments in the file, however. Note that as of revision 12, all DAM files are fully indexed so DAM files never go into SAM mode, so increasing the length of a segment does not slow down the file.

SIZE Function

(Line 36)	NUMBER OF ENTRIES:	(numeric)	(Line 37)
(Line 37)	INDEX:	(Number from 0-19)	(Returns to Line 23)
		(DATA)	(Returns to Line 23)
		(TOTAL)	(Returns to Line 23)

SIZE allows the user to specify an anticipated number of entries for a file. CREATK determines the number of segments and disk records required for any single index, the data subfile or the whole file.

If a number from 0 to 19 is specified only that index is displayed. The following information is provided:

1. Number of disk blocks (440 and 1024 word)
2. Number of segments required to contain index blocks
3. Number of segments allocated for the index

For the data subfile the following information is provided:

1. Number of disk blocks (440 and 1024 word)
2. Number of segments required to contain index blocks
3. Number of segments allocated for the index

For the total file the following information is provided:

1. Number of disk blocks (440 and 1024 word)
2. Number of segments required to contain index blocks
3. Number of segments allocated for each index subfile
4. Number of segments allocated for the data subfile
5. Total of all index subfiles and the data subfile

SELECTING A VERSION OF CREATK

CREATK Options

CREATK has two optional features that are determined when it is built. Direct access support may be suppressed, and long indexes (for very large files) may be included. In either case, CREATK must be rebuilt with the appropriate command file to incorporate or remove the option. As delivered, CREATK is created by the command file C←CREA and supports direct access but does not support long indexes.

Direct Access Support

In general, Direct Access support adds about 300 words to any MIDAS applications program that includes it. If the user decides that this support is unnecessary, KIDALB and/or VKDALB may be rebuilt and CREATK rebuilt with either C←CREA (no long indices) or C←LCRE (with long indices).

Long Indexes

Users with very large files may find that there is not enough room in some index subfiles to contain index entries for all data entries to be indexed. This can be overcome by permitting indexes requiring more space to use two adjacent index positions, i.e., to create a long index. The implications of a long index are that one potential index subfile is lost for each index created as a long index. For example, if index-subfile-0 is declared to be a long index, then the first available secondary index is index-subfile-2.

For MIDAS Rev. 14 and later, the size of an existing index subfile may be increased by increasing the length of the segments making up the index and REMAKEing the file with this new length. (See Section 4 for further information). Other users must use the long index feature as described below.

If the user discovers that the default index-subfile size is too small, after a file already has data in it, the long index feature may be invoked on that index-subfile if the next available position in the MIDAS file segment directory index-subfile is free. If the next index-subfile position is not free, one solution is to move the affected index-subfile further down the list of secondary-indices where two index-subfile positions are available and to recode applications using the index-subfile position to reflect the new index number requested by CREATK.

To clarify this concept, consider as an example that secondary-index-1 is running out of room and there is no secondary-index-2. The solution would be to invoke the long index version of CREATK on the file and to modify the description of secondary-index-1 to make it a long index. On the other hand, suppose there are secondary-indexes 2, 3, and 4. The only possibility in this case is to write a program to build a new secondary-index-5 (created as a long index) from old secondary-index-1.

All programs using secondary-index-1 have to modified to use secondary-index-5.

CREATK DEFAULTS

At Rev. 14, some of the default values for CREATK enhance support for storage modules. In particular, the default size for an index block has been changed to 1024 words, the length of the segment directory has been changed to 512 segments and the length of a DAM file to 512 disk records. Users with no storage modules who prefer the other defaults can easily build a suitable MIDAS by modifying the parameter file KPARAM. In any case, this does not affect existing MIDAS files. They continue with the same block sizes, etc. that they were created with. To convert existing files to new default block sizes, run each index through the MODIFY path of CREATK and respond to the BLOCK SIZE query with a (CR).

SECTION 3

KBUILD UTILITY

KBUILD FUNCTIONS

KBUILD is a utility that may be used to build a MIDAS file from one or more sequential disk files. KBUILD may also be used to add data records to an existing file.

The MIDAS file created by KBUILD will have no index information in overflow upon completion of the run.

The incoming data may be unsorted, or sorted on the primary key and/or one or more secondary keys. The user is asked if, and how, the input data is stored, and KBUILD makes use of this information to build the file more efficiently. When adding to an existing MIDAS file, the input file must be declared to be unsorted.

Note

> The program KBUILD cannot be used to process variable-length data, either as input information or for creation of a variable-length data MIDAS file. Use the MIDAS subroutine BILD$R, or PRIBLD and SECBLD, to process variable-length data.

KBUILD can build secondary-index-subfiles with no secondary data in the associated data record. For detailed information, refer to "Specifying Secondary Indexes" in this Section.

Keys associated with the secondary indexes (secondary keys) do not have to be included in the data entries. To exclude secondary keys, they must occur following the data portion of the input data entry. Secondary keys are then truncated from the end of the entry, before the entry is written to the MIDAS file.

If only secondary indexes are to be built, the user has the choice of adding one or more secondary indexes either from information contained in the MIDAS file itself, or of providing the required keys from a sequential disk file.

Existing data records cannot be modified with KBUILD. However, new records may be added to an existing file with KBUILD. The technique to do this is discussed in the latter part of this section.

KBUILD always builds a MIDAS file with at least one secondary index record in each secondary index subfile for each data entry; i.e., as each data entry is written, a secondary index record is also written to each secondary index subfile. However, it is not necessary to build all secondary indexes with KBUILD. A sparse secondary index can be built later, either with a user program or by KBUILD using a sparse input file. ("Sparse" implies fewer index entries than data entries.)

KBUILD DIALOG

Following is a step-by-step walk-through of the KBUILD dialog. To assist the user in following the dialog, line numbers have been assigned to each step. For every step requiring user response, all possible answers are shown. A "goes to" statement in parentheses, following each response, indicates where the dialog resumes. The entire dialog is summarized in Figures 3-1 through 3-3.

To commence KBUILD, the user types:

```
KBUILD
```

at PRIMOS command level. KBUILD then returns with the first question of the dialog as follows:

(Line 1)	SECONDARIES ONLY?	YES	(Goes to Line 2)
		NO	(Goes to Line 4)
		(OTHER)	(Repeats Line 1)
(Line 2)	USE KI/ DA DATA ENTRIES?	YES	(Goes to Line 3)
		NO	(Goes to Line 4)
		(OTHER)	(Repeats Line 2)
(Line 3)	ENTER KI/DA FILENAME:	FILENAME	(Goes to Line 10)
(Line 4)	ENTER INPUT FILENAME:	FILENAME OR TREENAME	(Goes to Line 5)
		(OTHER)	(KBUILD aborts)

CAUTION

Invalid responses in lines 4, 5 and 8 will cause KBUILD to abort. The user will be returned to PRIMOS command level.

(Line 5)	ENTER INPUT RECORD LENGTH:	(WORDS)	(Goes to Line 6)
		(NUMERIC=512)	(Goes to Line 6)
		(OTHER)	(KBUILD aborts)

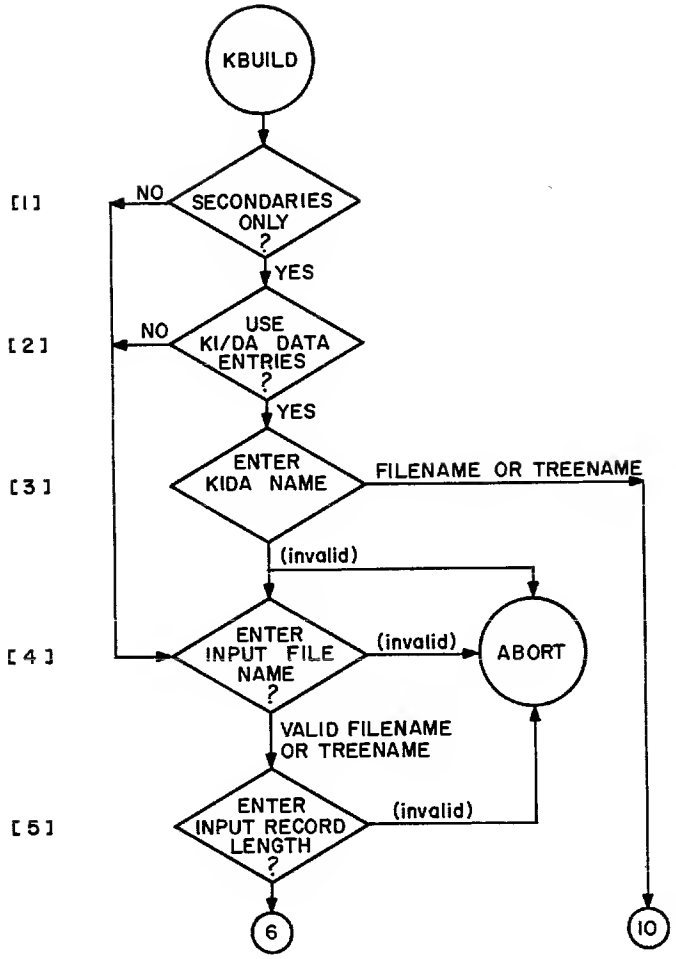


Figure 3-1. KBUILD Dialog (Lines 1-5)

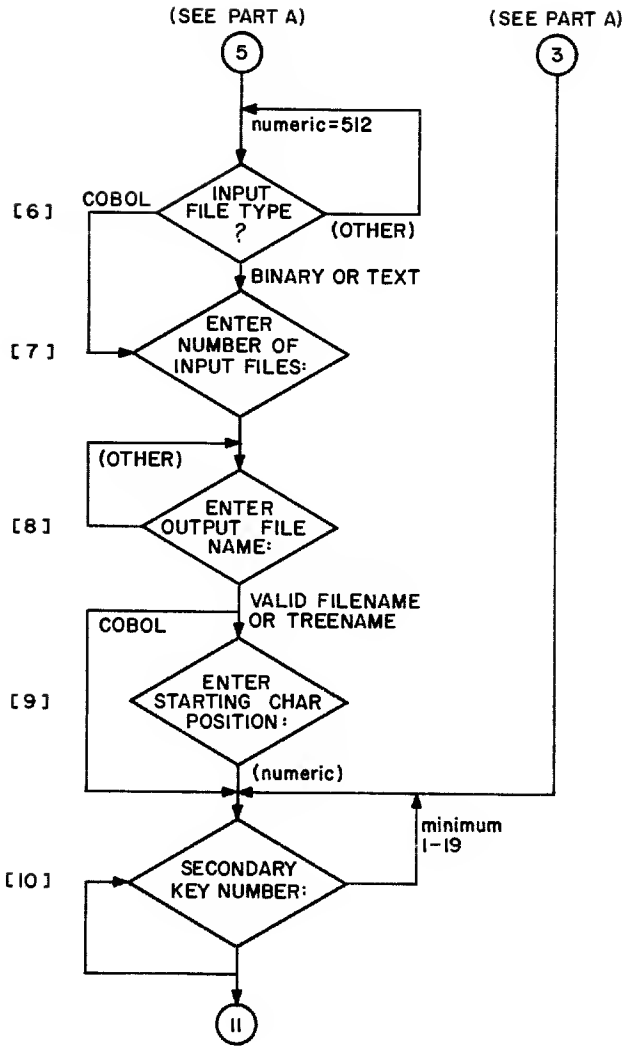


Figure 3-2. KBUILD Dialog (Lines 6-10)

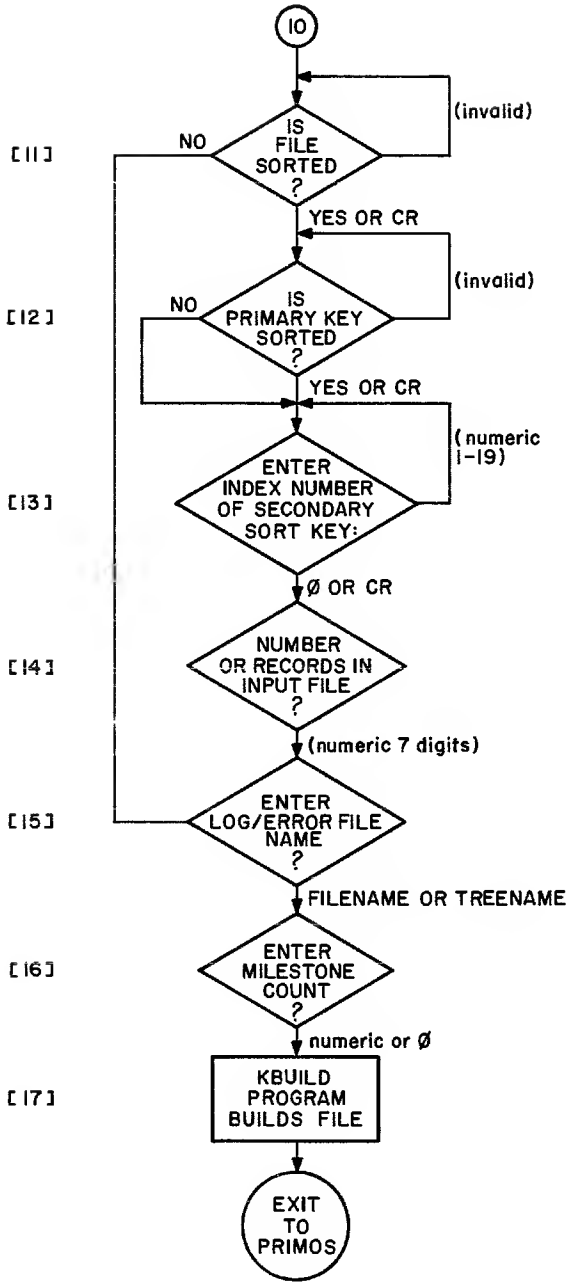


Figure 3-3. KBUILD Dialog Conclusion (Lines 11-18)

(Line 6)	INPUT FILE TYPE:	(T OR TEXT)	(Goes to Line 7)
		(C OR COBOL)	(Goes to Line 7)
		(B OR BINARY)	(Goes to Line 7)
		(OTHER)	(Repeats Line 6)
(Line 7)	ENTER NUMBER OF FILES:	(NUMERIC)	(Goes to Line 8)
(Line 8)	ENTER OUTPUT FILENAME:	FILENAME OR TREENAME	(Goes to Line 9)
		(OTHER)	(KBUILD aborts)
(Line 9)	ENTER STARTING CHARACTER POSITION, PRIMARY KEY:	(NUMERIC)	(Goes to Line 10)
(Line 10)	SECONDARY KEY NUMBER:	(NUMBER 1 to 19)	(Goes to Line 11)
		(Ø or CR)	(Goes to Line 12)
(Line 11)	ENTER STARTING CHARACTER POSITION:	(NUMERIC)	(Goes to Line 10)
(Line 12)	IS FILE SORTED?	YES OR CR	(Goes to Line 13)
		NO	(Goes to Line 16)
		(OTHER)	(Repeats Line 12)
(Line 13)	IS THE PRIMARY KEY SORTED?	YES OR CR	(Goes to Line 14)
		NO	(Goes to Line 14)
		(OTHER)	(Repeats Line 13)
(Line 14)	ENTER INDEX NUMBER OF SECONDARY SORT KEY:	(NUMERIC)	(Loops back to Line 13)
		(Ø OR CR)	(Goes to Line 15)

The loop will continue until the user responds to line 14 with either a zero or a CR. This signifies the last index.

(Line 15)	NUMBER OF RECORDS IN INPUT FILE:	(NUMERIC 7 DIGITS)	(Goes to Line 16)
(Line 16)	ENTER LOG ERROR FILENAME:	FILENAME OR TREENAME	(Goes to Line 17)
(Line 17)	ENTER MILESTONE COUNT:	(NUMERIC, or Ø)	(Exits)

Following line 17 the program exits to the KBUILD file building program.

INPUT FILES

The following paragraphs describe the contents of input files to KBUILD and give example input file formats for use with FORTRAN and COBOL. The input stream to KBUILD may come from one or more disk files. The file names may be supplied to KBUILD as a treename. Multiple input files are processed sequentially, one at a time.

If more than one input file is to be processed, the name of the first file must end in two digits and additonal files must have the same name, but with the number at the end incremented. For example, to use three input files in a UFD named BUILD, the first of the input files could be called INPØ1, the second file would be called INPØ2, and the third INPØ3. If the user is not attached to the UFD names BUILD, then the name specified to KBUILD to identify all three files would be BUILD>INPØ1. File names may be more than six characters on new style PRIMOS file system partitions. (Rev. 14 and later.)

Sorted Input Files

Input data may be sorted on one or more key fields, one of which may be the primary key. At run time, the user specifies whether the file is sorted. If so, faster routines can be used to build the indexes (including the primary index) corresponding to the sorted key fields. An example of a file with sorted key fields is a file that was being built with a second copy of the primary index as a secondary index for security reasons. If the incoming data were sorted on a primary key, then it is also sorted on the key of the duplicate index. If the file is declared sorted, the user is asked to supply the numbers of the keys on which the file is sorted and the number of entries in the input file.

Note

Even if the input data is sorted, if KBUILD is being used to add to a file the input file must be declared unsorted. If the file is declared sorted, all index subfiles that correspond to the sorted keys contain only the new records.

Format of Input Data Files

KBUILD processes input files that are either in text file format (TEXT), binary format (BINARY), or COBOL format (COBOL).

In TEXT format, the data is entirely in character form. The input file may be created, examined or modified by the PRIMOS Text Editor. Text files are created by FORTRAN WRITE statements and COBOL programs.

A BINARY file may contain text information, but the end of a record is not determined by the presence of a new-line character. Other data types, such as integer or floating point numbers, can be included in the data record. A binary file can not normally be examined by the Editor. Binary files can be created using the PRIMOS file system subroutines PRWFIL or PRWF$$. They can be accessed by FORTRAN programs but not by COBOL or RPG programs.

COBOL data files must be in TEXT format but with the primary key as the first field in the data entry.

Format of Input Data Records

The input record must be already formatted for placement in the MIDAS file. The data portion of the MIDAS record must be the first part of the input data record. The primary key may be included in the data portion of the MIDAS record. COBOL requires this and also requires that the primary key be the first field in the MIDAS data entry. Ordinarily, however, it is not necessary for the user to keep a separate copy of the primary key since MIDAS keeps a copy with the data and can return its copy to the user upon request. The primary key must be included in the input data record. If not a part of the user's data, the primary key must follow in the area of the input data record that will not be written out to the MIDAS file.

There are no further requirements for the format of the input data record, except that the portion to be included in the MIDAS file must be the exact image of the MIDAS data record, and the data in the input file must correspond to either TEXT or BINARY format.

Specifying Secondary Indexes

KBUILD can build secondary index-subfiles with no secondary data in the associated data record and in which there is a secondary index subfile for each data record. It is not necessary to build any or all secondary index subfiles with KBUILD. At run time, the user specifies which index subfiles are to be built and the byte offset (character position) within the input data record of the secondary key field. To indicate that there are no secondary index subfiles or that the last index has been processed, a carriage return (CR) is entered.

If the record is in binary format, the format of the data record must be arranged so that all key values begin on a byte boundary (i.e. key may begin either on Bit 1 or Bit 9 of a word). This happens automatically if the record is in text format. If some secondary keys are to be excluded from the data in the MIDAS file, secondary keys must be placed at the end of the input data record so that they may be truncated from the record when it is written to the MIDAS file.

Input Record Length

The input record length is the length of the data entry in full as it resides in the input file. The input record length must include the length of the primary key, the length of the user's data, and the length of any secondary keys not to be included in the data. Since KBUILD only builds MIDAS files with fixed-length data entries, it is not necessary to supply KBUILD with the length of the MIDAS record. MIDAS retains this information in the file. Consequently, KBUILD is able to truncate the input data to its correct length without the user supplying this length.

At run time, the input record length is specified in computer words (two characters per computer word).

COBOL INPUT FILE EXAMPLE

Assume that COBOL user wishes to build a MIDAS file that has a 128 character (64 word) data entry. There will be four secondary index subfiles, specified by four keys, three of which are parts of the primary key, which is 29 bytes long. The offsets of these key fields are:

Byte 1 - Primary key - as required by COBOL
Byte 2 - Secondary index 1, key length 28 bytes
Byte 2 - Secondary index 2, key length 6 bytes
Byte 9 - Secondary index 3, key length 6 bytes

Note that these secondary keys are portions of the primary key.

The fourth secondary key begins on byte 52. This key is ten characters (bytes) long. Since MIDAS keeps track of the length of each of the keys, the key lengths do not have to be supplied to KBUILD. The data fields in the record that are not key fields are of no interest to MIDAS: therefore, they are not described to KBUILD. The user specifies COBOL format to KBUILD, and all 64 words in the input data entry are written to the MIDAS file.

FORTRAN INPUT FILE EXAMPLE

Assume that another user wishes to build a MIDAS file to be accessed only by FORTRAN applications. The data entry, not including the primary key, will be 50 words long. Most of the information in the file will be numeric fields that can be stored in floating point or long integer format. The user has converted his data from its original source and prepared an input data file. An input data record is 63 words long and has the following format:

data	words 1 through 50
primary-key	words 51 through 61; key type is character. The user will use the MIDAS copy when retrieving the data record so the key will not be included in the user's data record.
secondary-key-1	words 62 and 63; key type is long integer and is not required as part of the data entry.
secondary-key-3	words 1 through 4; key type is double precision floating point and is part of the data.

There is a second secondary index, but the user does not require an entry in this index for every data record. He has decided to have the values for this index entered as needed by the interactive applications program that will normally process the file. KBUILD will write the 50 data words to the MIDAS file. Words 51 through 63, which contain key information not to be included in the data, will not be included. At run time, the user will specify the input file type to be BINARY.

These examples are further explained later in this section.

OUTPUT FILE

The specified output file must be a MIDAS file. If a new file is to be built, these must be a template file created by CREATK. When building a new file, use of an old MIDAS file (with data already in it) as the output file results in a broken file containing some of the old information as well as the new.

When specifying the name of a MIDAS output file, a treename may be used and on new partitions, long names are also acceptable.

REPORT AND ERROR FILE

KBUILD reports nonfatal errors and milestone timings to the user both on the user's terminal and to a og/ rror file. The user supplies the name of this file at run time. The name may be a treename and may be up to 32 characters long on a new partition.

Milestone Reports

At run time, the user enters a milestone count that is used as a counter for printing a milestone time record. For example, if the user enters a milestone count of 100 then every one hundred records, a milestone time record is printed for the user's terminal and written to the Report/Error file. If a milestone count of 0 is entered, milestone records are only printed for the first and last records.

A sample milestone report for building a MIDAS file from an input file named FILE01, containing 103 records, with a milestone count of 50, is shown as follows:

```
FILE01
COUNT       DATE       TIME    CPU MIN DISK MIN  TOTAL TM   DIFF

    0     05-23-77   12:34:52  0.000    0.000     0.000    0.000
   50     05-23-77   12:38:06  0.384    0.179     0.563    0.279
  100     05-23-77   12:40:42  0.822    0.340     1.161    0.297
END OF RUN
  103     05-23-77   12:40:56  0.882    0.354     1.236    0.074
```

The name of the input file FILE01 appears as the first line in the report. This is followed by column headers as follows:

Column	Contents
COUNT	the number of entries processed (up to $2^{31}-1$)
DATE and TIME	date and time of entry (time hrs., mins., secs.)
CPU MIN	minutes of CPU time used since start of run
DISK MIN	minutes of disk I/O time (including paging) from start
TOTAL TM	sum of disk and CPU time since start
DIFF	increment in total time since last milestone

After the one-hundred-and-third record is processed, an end of file is encountered, and the final line is preceded by the note: END OF RUN.

Report Records

In addition to the milestone report, other conditions generate a report record and milestone type record. In particular:

In a milestone report, when opening a new input file, the new file name is printed, for example:

```
FILE 02
COUNT    DATE       TIME     CPU MIN  DISK MIN   TOTAL TM   DIFF
 103   05-23-77  12:40:56     0.882     0.354       1.236   0.074

FILE HANDLER ERROR  12  (FILHER)
2000725227737902763450000000000122 2NE0000099001176000000000DBLT000
RECORD NOT ADDED
  8    05-27-77  16:33:41     0.031     0.005       0.037   0.004
```

which indicates that the eighth record was not added because the entry was already in the file (error code 12).

In a milestone report, when a secondary index entry is not added due to an error input, a record is printed plus other descriptive text, for example:

```
KEY SEQUENCE ERROR (KX$CHK)
20006775A1080402762400000000000013K0000009900160720000000DBLT000
SECONDARY KEY 02 NOT ADDED
  5    05-27-77  16:10:54     0.041     0.021       0.062   0.010
```

which indicates that an entry was not added, either because there was a duplicate occurance of a key in an index not supporting duplicate entries or than an entry out of order was detected in a sorted input system.

Fatal errors are written to the log/error file before KBUILD aborts.

COBOL Program Example

The offsets of the secondary keys were discussed above. In addition assume that there are three input files in the UFD named BUILD>INPØ1. INPØ2, and INPØ3. Assume that the input data is sorted on secondary key 2 (six bytes of the primary key beginning in byte 2) but not on key 1 (28 bytes beginning in the same location). In addition, among the three input files, there are 23000 data records to add. The file will be built into an existing MIDAS template file called ACCRC1. The KBUILD interactive session would be as follows (user input is underlined):

```
ENTER INPUT FILE NAME:          BUILD>INPØ1
ENTER INPUT RECORD LENGTH (WORDS): 64
INPUT FILE TYPE: COBOL
ENTER NUMBER OF INPUT FILES: 3
ENTER OUTPUT FILE NAME: ACCRC1
SECONDARY KEY NUMBER: 1
ENTER STARTING CHARACTER POSITION: 2
SECONDARY KEY NUMBER: 2
ENTER STARTING CHARACTER POSITION: 2
SECONDARY KEY NUMBER: 3
ENTER STARTING CHARACTER POSITION: 9
SECONDARY KEY NUMBER: 4
ENTER STARTING CHARACTER POSITION: 52
SECONDARY KEY NUMBER: (CR)
IS FILE SORTED? YES
IS THE PRIMARY KEY SORTED? NO
ENTER INDEX NUMBER OF SECONDARY SORT KEY: 2
ENTER INDEX NUMBER OF SECONDARY SORT KEY: (CR)
NUMBER OF RECORDS IN INPUT FILE: 23000
ENTER LOG/ERROR FILE NAME: ERR01
ENTER MILESTONE COUNT: 1000
```

Suppose that at some future time, an input file INPØ4 which has the same description as INPØ1, - 2, and -3 is to be added to the already existing MIDAS file. The KBUILD session to add to the file would be as follows:

```
ENTER INPUT FILE NAME:          BUILD>INPØ4
ENTER INPUT RECORD LENGTH (WORDS): 64
INPUT FILE TYPE: COBOL
ENTER NUMBER OF INPUT FILES: 1
ENTER OUTPUT FILE NAME: ACCRC1
SECONDARY KEY NUMBER: 1
ENTER STARTING CHARACTER POSITION: 2
SECONDARY KEY NUMBER: 2
ENTER STARTING CHARACTER POSITION: 2
SECONDARY KEY NUMBER: 3
ENTER STARTING CHARACTER POSITION: 9
SECONDARY KEY NUMBER: 4
```

```
ENTER STARTING CHARACTER POSITION:  52
SECONDARY KEY NUMBER:  (CR)
IS FILE SORTED?  NO
ENTER LOG/ERROR FILE NAME:  ERR02
ENTER MILESTONE COUNT:  1000
```

The FORTRAN Program Example

As additional information for the FORTRAN example described above, suppose the file is sorted on the primary key only, that there is one input file containing 10100 entries called FILE01 in the current UFD and that the output file is a MIDAS template file called CUSTFIL.KIDA which is on a new partition UFD called NEWPAR. The error file ERRFIL.KIDA will also be written to this UFD. The following is an example for FORTRAN (user input is underlined):

```
ENTER INPUT FILE NAME:  FILE01
ENTER INPUT RECORD LENGTH (WORDS):  63
INPUT FILE TYPE:  B
ENTER NUMBER OF INPUT FILES:  1
ENTER OUTPUT FILE NAME:  NEWPAR CUSFIL.KIDA
ENTER STARTING CHARACTER POSITION, PRIMARY KEY:  51
SECONDARY KEY NUMBER:  1
ENTER STARTING CHARACTER POSITION:  61
SECONDARY KEY NUMBER:  3
ENTER STARTING CHARACTER POSITION:  1
SECONDARY KEY NUMBER:  (CR)
IS FILE SORTED?  (CRLF)
IS THE PRIMARY KEY SORTED?  (CR)
ENTER INDEX NUMBER OF SECONDARY SORT KEY:  (CR)
NUMBER OF RECORDS IN INPUT FILE:  10100
ENTER LOG/ERROR FILE NAME:  NEWPAR ERRFIL.KIDA
ENTER MILESTONE COUNT:  (CRLF=0)
```

SECTION 4

REMAKE UTILITY

REMAKE FUNCTIONS

REMAKE is invoked to restructure a MIDAS file. Restructuring is required when the user has added or deleted a large number of data entries or secondary index entries, using ADD1$ or DELET$. New entries are added into overflow areas, and when these areas become large, searching becomes very inefficient. Furthermore, deleted entries represent empty space on the disk, and reduce the file storage efficiency.

REMAKE has several options. Individual secondary indices may be restructured, all indices may be restructured, all indices and the data may be restructured, or the whole file may be restructured and rewritten into a new file.

Restructuring an index causes the index to be built with all overflow areas incorporated into the main body of the index and causes space occupied by deleted index entries to be recovered. This is done within the existing file structure. REMAKE must have enough space available on the disk to make a temporary second copy of the largest index to be restructured. If REMAKE aborts due to insufficient disk space, the MIDAS utility KIDDEL (refer to Section 6) may be used to delete the partially built replacement index. The file may continue in use until sufficient disk space is available for REMAKE to run to completion.

When the data is restructured, all indexes are rebuilt, and in addition, the data subfile is rebuilt with the data entries put into the data subfile in sorted order on the primary key. Secondary indices must be restructured during a data REMAKE, otherwise, the pointers into the data subfile from these indices would be incorrect. Space occupied by deleted data entries is recovered. In a data restructure, REMAKE must have as many free disk records on the disk containing the file as are required by the data-subfile and the primary-index. If REMAKE aborts due to insufficient disk space while restructuring the data and primary-index, the file may be recovered, and the data REMAKE tried again when more disk space is available. Alternately, the file may be rebuilt into a new file on another disk.

If the file is rebuilt into a new file, REMAKE creates a new template for the file, if no template exists. If a template exists with the new name, it is used. If the new template differs from the old (for example if the length of the data entry is changed), the values in the new template are used to describe the new file. If the template is an old MIDAS file (with data already in it), REMAKE cleans out the old information before inserting the new.

When the restructure of a MIDAS file is complete, all modifications to the subfiles that have been implemented using CREATK (or other means) are incorporated into the restructured MIDAS file.

REMAKE DIALOG

Following is a step-by-step description of the REMAKE dialog. To assist the user in following the dialog, line numbers have been assigned to each step. For every step requiring user response, all the possible answers are shown. A "goes to" statement in parentheses, following each response, indicates where the dialog resumes.

To start REMAKE, type:

```
REMAKE
```

at PRIMOS command level. REMAKE returns with the first question of the dialog as follows:

(Line 1)	FILE NAME?	(treename or filename)	(Goes to Line 2)
		(other)	(Repeats Line 1)
(Line 2)	INDICES?	DATA	(Goes to Line 3)
		ALL	(Goes to Line 3)
		NEW	(Goes to Line 4)
		(1 to 20 numeric fields (0 to 19) separated by spaces)	(Goes to Line 3)
		(other)	(Repeats Line 2)

Typing DATA in line 2 causes REMAKE to recreate all index subfiles and the data. REMAKE requests verification before proceding as this procedure cannot be aborted. ALL orders the REMAKE of all defined indexes according to the existing data subfile. To REMAKE an existing file into a new file use NEW. The new file can be on another logical disk. To specify which index subfiles are to be recreated, type the number(s), separating each field by a space.

(Line 3)	DONE EXIT		(Returns to PRIMOS)
(Line 4)	NEW FILE NAME?	(filename or treename)	(Goes back to Line 3)
		(other)	(Repeats Line 4)

When the specified recreation(s) are complete, REMAKE displays DONE on the user's terminal and returns to PRIMOS.

Notes

1. The response D or DATA causes REMAKE to ask the user to verify that he wishes to continue. Upon completion of the procedure, all index subfiles and the data have been recreated. The user may not abort this procedure.

2. The response A or ALL results in recreation of all the defined indices according to the existing data subfile.

3. REMAKE restructures the entire existing file with the new name. The file may be on another logical disk.

4. One or more numeric fields cause only the index subfiles specified to be recreated.

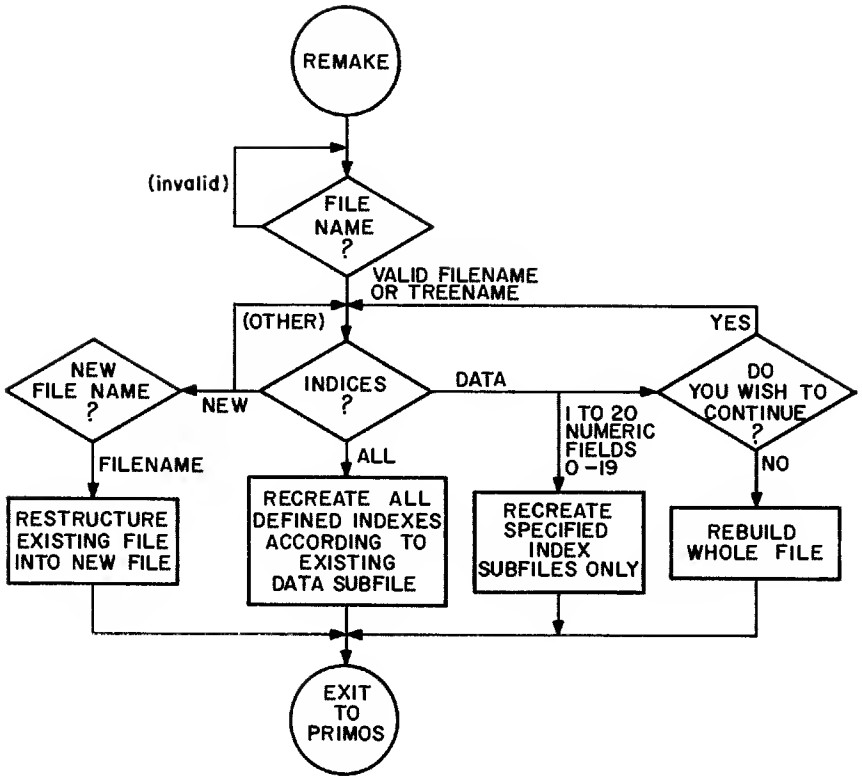


Figure 4-1. REMAKE Dialog

SECTION 5

REPAIR UTILITY

REPAIR is a utility module that may be used in an attempt to repair a MIDAS file that has been damaged by a system or MIDAS program crash. The REPAIR program constructs a new MIDAS file from any portions of the damaged file that can be interpreted correctly.

There is no guarantee that REPAIR can completely restore a damaged file. It cannot recover file damage resulting from a disk failure (or user program failure) that has overwritten, deleted or removed information from a file by truncation (e.g., by FIXRAT). In some instances, most of the file is usable, but because of the damaged section, cannot be accessed through the MIDAS routines or utilities. When a problem such as this exists, REPAIR prints a message at the user's terminal (and into a file) indicating where data is missing from the damaged file. However, REPAIR can recover the usable portions of the damaged file.

At run-time, the user may choose to attempt to recover the whole file (the primary index subfile plus data and all secondary index subfiles), or just the primary index subfile and data plus selected secondary index subfiles. This option is useful if alternative means are available for creating secondary index subfiles, and it is prefered to insure that no secondary index entries are missing from data entries recovered.

PRINCIPLES OF REPAIR OPERATION

REPAIR requires one input file (the damaged MIDAS file), and two output files: a MIDAS template file and an error logging file. There is no assurance that the template portion of a damaged file is intact. Therefore, before using REPAIR, you must run CREATK and generate a new template file identical to the original template. (Just run your original CREATK dialog command file.)

REPAIR asks for the physical disk record length (440 or 1024 words) so that it has the best chance to handle defective disk records. REPAIR also allows the user to specify which secondary index subfiles are to be recovered. REPAIR requires sufficient disk space of the same or another disk to build a new copy of the file. After the initial dialog, the REPAIR program constructs a new MIDAS file from all portions of the damaged file that can be correctly intepreted. Valid data entries which simply cannot be accessed by MIDAS are recovered. Overwritten, deleted, or truncated data cannot be retrieved. REPAIR does not use the normal data access routines, and therefore is able to retrieve information that is no longer accessible through MIDAS routines and utilities. REPAIR builds a new MIDAS file using the new template and what exists of the old file.

REPAIR is a multi-pass program operating on the MIDAS subfiles in this order:

- Primary Index
- Primary Index Overflow
- Data Entries
- Secondary Index 1
- Secondary Index 1 overflow
- Last secondary Index
- Last secondary Index overflow

The purpose of this approach is to recover as much as can be retrieved. The user is offered the option of recovering any or all secondary index subfiles at the time of repair, or using another program or KBUILD to recover secondary index information.

THE REPAIR DIALOG

Following is a detailed description of the REPAIR dialog. It is summarized in Figure 5-1.

(Line 0)	REPAIR		(Goes to Line 1)
(Line 1)	OLD FILE NAME:	(filename or treename) (other)	(Goes to Line 2) Exit to PRIMOS (abort)

The only restriction on the source file is that it must have been created by PRIME software Revision 14. (or later) or have been processed by Rev. 14. (or later) REMAKE to rebuild all indexes. The name of the damaged file may be supplied as a treename if it is not in the home UFD.

CAUTION

Supplying an invalud response in lines 1 and 3 will cause REAPIR to abort. User is returned to PRIMOS command level.

(Line 2)	ENTER LOG ERROR FILE NAME:	(filename or treename)	(Goes to Line 3)

The user is asked to supply the name of the error logging file. As usual the name may be a treename.

(Line 3)	NEW FILE NAME:	(filename or treename) (other)	(Goes to Line 4) Exit to PRIMOS (abort)

The new template file must correspond in every respect to the original template from which the old file was built. It need not reside in the same UFD as the old file nor on the same logical or physical disk as the old file. The user may supply a treename to identify the file.

(Line 4)	SECONDARY INDICES:	ALL	(Goes to Line 5)
		(number 1 to 19)	(Goes to Line 5)
		(CR)	(Goes to Line 5)

In line 4 if the response is ALL, then all defined indexes will be recovered. When a number (1 to 19) is specified only those indexes will be recovered. Typing a CR indicates that no secondary indexes are to be recovered.

(Line 5) PHYSICAL DISK RECORD LENGTH: (numeric) (Goes to Line 6)

The number is either 1024 words for storage modules or 440 words for all others.

(Line 6) EXIT

The program exits to perform the REPAIR operation, i.e., restructure the MIDAS file. Refer to the discussion on action of REPAIR for a full explanation.

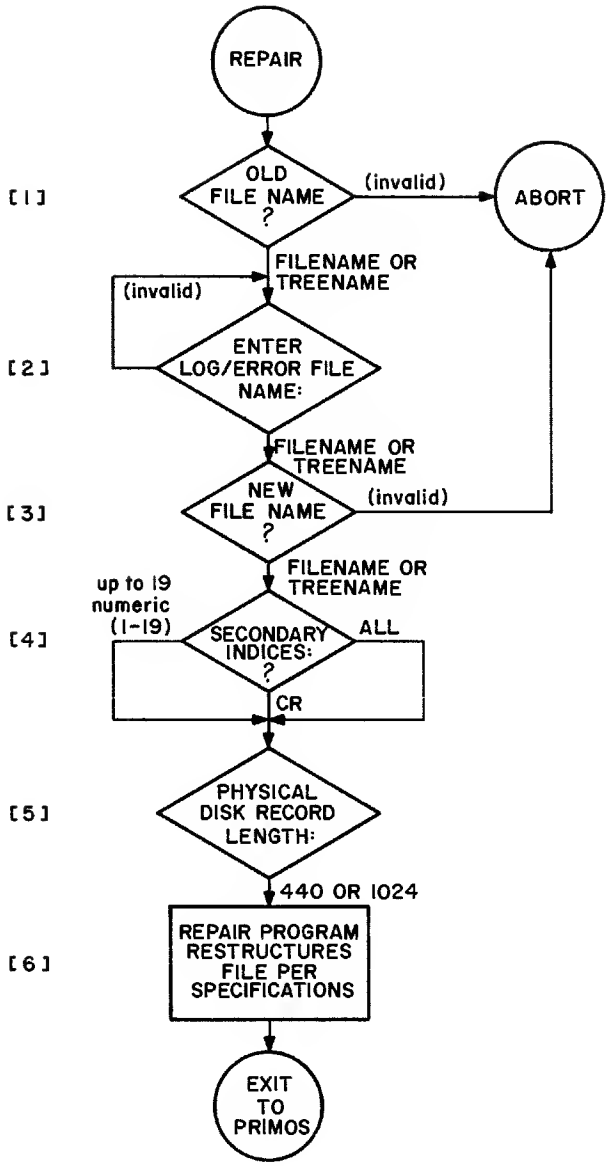


Figure 5-1. REPAIR Dialog

ACTION OF REPAIR

Method of File Recovery

REPAIR requires that the user supply a template file (generated by CREATK) corresponding to the file to be regenerated. All useful information found in the old file is put into the new, as specified by the user. The primary index and data are processed first. Then, a pass is made through each secondary index specified by the user to create that index in the new file.

Only entries in the old index corresponding to data entries in the new file are transferred to the new index.

Primary Index and Data

The primary index and data subfiles are considered to consist of three parts. First, there is the main body of the primary index which is composed of all entries indexed as of the last REMAKE. Second, there is the collection of entries added into overflow since the last REMAKE. Finally, there is the data itself. REPAIR traverses each of these three parts of the file to recover as much information as possible.

First, the main body of the primary index is examined and all entries for which there is a valid index entry and data entry are written to the new file. Entries where the information in the index and data subfiles either do not agree or are missing are noted in the error logging file. At the end of this pass, the message END PASS 1 is given.

Second, the overflow area of the primary index is examined in the same fashion as the main body of the index. At the end of this pass, the message END PASS 2 is given.

Finally, the data subfile is examined to recover any data records for which index entries have been lost. At the end of this pass, the message PRIMARY INDEX AND DATA COMPLETE is given.

As a result of this three-pass approach, the user has the best possible chance of not only recovering all the data, but also being the most informed about any missing entries.

Secondary Index Subfiles

Only those secondary index subiles specified by the user are processed. (See line 4 of dialog.) Secondary index subfiles are processed in two passes, corresponding to the main index and overflow passes of the primary index and data rebuild. Those secondary entries for which there is no longer a data entry available are flagged in the error file and on the user's terminal. All other entries are transferred to the new file.

Error Detection

REPAIR tries to open and read each segment in the segment director of the MIDAS file. In most cases, an index subfile does not use all the segments allocated for it. Similarly, the data subfile normally does not use all its segments. Nevertheless, one of the possible ways in which a MIDAS file can be damaged is that a segment is deleted, or truncated. For this reason, all segments that do not exist in an index subfile are reported as missing, though this does not mean that there is necessarily anything wrong. Additionally, the data subfile segments are read and reported as missing until twenty consecutive segments have been found missing. At this point, REPAIR assumes that there are no more data segments.

When an error in the file is encountered, an explanatory error message is printed. Each error message is flagged with the nearest entry that was successfully processed. When an error has been detected and REPAIR is able to begin processing again, the first entry successfully processed is also reported. This information gives the user some idea of where the problem areas are.

EXAMPLE OF REPAIR ACTION

As an example of the action of REPAIR, assume that the following REPAIR dialog takes place (user input is underlined). The old file has secondary indexes but the user does not wish to recover them:

```
OK, REPAIR
OLD FILE NAME: ACCR1                          Damaged file
ENTER LOG/ERROR FILE NAME: ERR01              In current UFD
NEW FILE NAME: ACCRC2                         In current UFD
SECONDARY INDICES: (CR)                       None
PHYSICAL DISK RECORD LENGTH: 1024             A storage module
. . .                                         Exit to REPAIR program
                                              processing
```

The dialogue results in the following error log in file ERR01:

PASS 1 - MAIN PRIMARY INDEX (REPAIR)	Remarks
ENTRY PROCESSED	Note 1
00211508A1120600743470000000 BYPASSED BAD DATA ENTRY (GETARC)	Note 2
00571505A10904000000000000000 ENTRY PROCESSED	
10063156280264017634300000000 LAST ENTRY PROCESSED MISSING SEGMENT (HUNT)	Note 3
10063156280264017634300000000 LAST ENTRY PROCESSED END PASS 1 (REPAIR) PASS 2 - OVERFLOW (REPAIR)	Note 4
100663156280264017634300000000 ENTRY PROCESSED	Note 5
10071282271187017630900000000 LAST ENTRY PROCESSED MISSING SEGMENT (HUNT)	
10071282271187017630900000000 LAST ENTRY PROCESSED MISSING SEGMENT (HUNT)	
10071282271187017630900000000 LAST ENTRY PROCESSED PRIMARY INDEX AND DATA COMPLETE (REPAIR)	

Notes - Error Log Example

1. The first entry in any MIDAS file is usually a dummy key that may be unprintable. This indicates that REPAIR has started.

2. There was an index entry for this key, but the data entry was unreadable. It was bypassed. REPAIR then reports on the next entry successfully processed.

3. This is the last entry in the main portion of the file. There is no overflow in this file so at this time REPAIR traverses all the missing segments, reporting on them as it goes. This is all right.

4. REPAIR has detected the end of the index, hence will close pass one. It then moves on to overflow, but there is none, so REPAIR moves immediately to the last pass.

5. There are some entries in the data file for which index entries have been lost. The first one of these has the key value 10063388 . . . the last one is also flagged and once again missing segments are processed, this time up to twenty of them. Finally, REPAIR notes that the primary index and data are rebuilt.

SECTION 6

KIDDEL UTILITY

KIDDEL FUNCTIONS

KIDDEL allows the user to delte an index subfile, the entire MIDAS file, or any unwanted segments that are left over form a non-fatal abort from a MIDAS subroutine. The dialog is described below and summarized in Figure 6-1.

KIDDEL DIALOG

(Line 1)	FILE NAME?	(filename or treename)	(Goes to Line 2)
		(other)	(Repeats Line 1)
(Line 2)	INDICES?	JUNK	(Goes to Line 3)
		ALL	(Goes to Line 3)
		(numeric 0-19, up to twenty fields separated by commas)	(Goes to Line 3)
		(other)	(Repeats Line 2)
(Line 3)	DONE		(Exits to PRIMOS)

J or JUNK reclaims unwanted segments at the end of the file. These segments are created by off-line procedures when building or rebuilding indices. If a procedure has aborted without destroying the file, the J feature of KIDDEL may be used to delete them.

ALL will result in the deletion of the entire file.

Specification of one or more numeric fields causes the specified index subfiles to be deleted. When the requested deletions have taken place KIDDEL returns to PRIMOS command level.

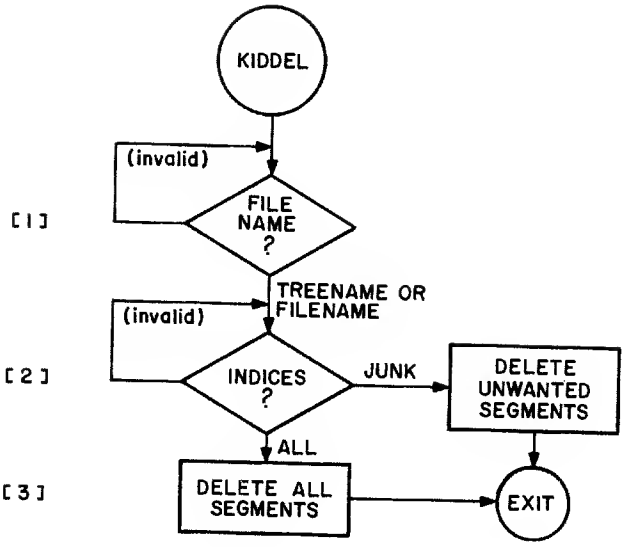


Figure 6-1. KIDDEL Dialog

SECTION 7

SUBROUTINES TO BUILD A MIDAS FILE (PRIBLD, SECBLD, AND BILD$R)

BUILDING FILES

Once source data and a template file are available, user maintenance routines may be used to build a MIDAS file (refer to Figure 1-1). One of these routines, BILD$R, accepts unsorted data and builds the file in overflow. Periodically, the index subfile and overflow are merged to create a new index subfile, so that the overflow is not allowed to become too large. Therefore, the process of building a MIDAS file runs more efficiently.

A second routine, PRIBLD runs much faster than BILD$R, provided that the source data is sorted by primary key.

There is also a routine SECBLD that may be used to build secondary indices from data sorted on a secondary key value.

Modifying Files

BILD$R may be used to add new entries to an existing file from either sorted or unsorted source data. PRIBLD and SECBLD may not be used to add entries to the file. PRIBLD, SECBLD and BILD$R may all be used concurrently to build a MIDAS file.

Communication Flag

All three of the single-user maintenance routines use a communication flag to control the flow of records through the routine. The values are:

Value	Meaning
0	Set initially to 0 by the user to indicate that the first record is to be processed.
1	Set by the called routine to indicate that the first entry has been processed. The value remains 1 until . . .
2	Set by the user after the last entry has been processed
3	Set by the called routine to indicate that the index has been finished and closed.

The single-user maintenance routines are serially reusable and the process may be started over for a new index or a new MIDAS file by setting the flag (SEQFLG) to 0 again.

The following paragraphs describe the calls to the maintenance routines BILD$R, PRIBLD and SECBLD. Variables or arrays specified in these calls must be specified as INTEGER*2 in FORTRAN programs that call these routines, with the exception of the parameter RNUM. RNUM specifies the number of expected entries to be added to the file, and it is defined as REAL to allow a MIDAS file to contain more than 32,000 entries.

In all cases, the user must open the template and the partially filled MIDAS file prior to the first call to any of these routines and must also open the file descriptor segment on the PRIMOS file unit assigned to ISSAM by a call to FILSET.

MAINTENANCE SUBROUTINE DESCRIPTIONS

PRIBLD

PRIBLD is a subroutine to build the primary index and data subfile from sorted source data. Calls to PRIBLD may be used concurrently with BILD$R and SECBLD. Before calling PRIBLD, the user must have assigned PRIMOS file unit numbers to ISUNIT, ISSAM, and ISDAT. Programs calling PRIBLD must be compiled with the $INSERT file OFFCOM from UFD named SYSCOM (i.e., $INSERT SYSCOM > OFFCOM). Also, the MIDAS file description subfile (template)must exist, and it must be open for reading on file unit ISUNIT. PRIBLD initially reads a new copy of the file descriptor subfile; therefore, the user should write out a previous copy of his own with the routine KX$BWT, if modifications have been made to the subfile prior to the user's first call to PRIBLD.

The calling sequence for PRIBLD is:

```
CALL PRIBLD (seqflg, keybuf, pbuffr, plngth, altrtn, rnum)
```

The Parameters have the following significance:

Parameter	Meaning
seqflg	Is the communication flag. Its possible values have been discussed. In summary they are 0 before first call to PRIBLD. 1 during operation. 2 & 3 at the time of last call to PRIBLD.
keybuf	Is a numeric variable or a one-dimensional array containing the current value of the primary key.
pbuffr	Is a one-dimensional array containing the data to be added. If plngth is zero, pbuffr may also be zero.
plngth	Contains the number of words of data supplied in pbuffr. If plngth is 0, the entry written to the MIDAS file is all zero's. If plngth is less than the length of pbuffr, the entry is zero-filled.
altrtn	Is an alternate return to be taken if an error occurs. If altrtn is zero, the routine returns to PRIMOS command level. The cause of the error will be printed at the user's command.
rnum	Is, approximately, the number of entries in the finished MIDAS file being built by PRIBLD. Rnum must be equal to or larger than the number of entries in the file. If rnum is 0, or not supplied, the default value assumed by PRIBLD is 200,000. rnum is a real variable.

SECBLD

SECBLD is a routine to build a secondary index from sorted data. The secondary keys must be in sorted order.

SECBLD may be used to build several secondary index subfiles concurrently with the primary index and source data. SECBLD may be used in conjunction with PRIBLD or BILD$R or by itself. SECBLD may not be used to add new index entries to an existing index.

Programs that call SECBLD must be compiled with the $INSERT file named OFFCOM in the UFD named SYSCOM on the master disk(i.e., $INSERT SYSCOM> OFFCOM). Before calling SECBLD, the user must have assigned PRIMOS file unit numbers to ISUNIT, ISSAM, ISDAM, and ISDAT. The MIDAS file description subfile (template) must also exist, and it must be open for reading on the file unit specified by ISUNIT. A call to the routine FILSET must be made to read the file descriptor. In addition, the flag word in array JTEMPS must be set to zero.

When all entries have been added, a final call is made to SECBLD, for each index, with the communication flag set to 2. If this final calling sequence is not made, the index is not useable.

Calling Sequence

The calling sequence for SECBLD is:

```
CALL SECBLD (jtemps, keybuf, prikey, index, pbuffr, plenth,
             altrtn, rnum)
```

The meaning of the parameters is as follows:

Parameter	Meaning
jtemps	Is a three-word flag and scratch area that permits SECBLD to run concurrently with itself, PRIBLD and BILD$R. jtemps (1) is the communication flag for SECBLD. The remaining words, jtemps (2) and jtemps (3), are used internally by SECBLD and must not be modified by the user. If SECBLD is being used to build several index subfiles concurrently, a separate copy of jtemps is required for each index being built.
keybuf	Is a one-dimensional array that contains the current value of the secondary key.
prikey	Is a one-dimensional array that contains the value of the primary key for this entry.
index	Is the number of the secondary index to which the entry is to be added (1 through 19).
pbuffr	Is a one-dimensional array that contains any secondary data to be added. If plenth is zero, pbuffr may be zero also.
plenth	Is the length of the data supplied by the user. If the plenth is zero, or less than the MIDAS file handler expects, the user data entry to be written to the file is zero-filled. If no user data is expected, none will be added to the file.
altrtn	Is an alternate return to be taken if an error occurs. If an error occurs and altrtn is zero, the program aborts and exits to PRIMOS. The cause of the error will be indicated at the user's terminal.

rnum — Is approximately the number of entries to be added to the file by SECBLD. Rnum must be equal to or greater than the number of entries in the file. If rnum is Ø, the default value is 200,000. rnum is a REAL variable.

BILD$R

BILD$R is a MIDAS single-user maintenance subroutine that can be used to build a keyed-index or direct access MIDAS file from unsorted data. BILD$R adds an entry to the overflow area of an established MIDAS file. BILD$R may also be used to add new entries to an existing file or index. The arguments are set up as for their equivalents in the calling sequence of ADD1$. The advantage of BILD$R over ADD1$ is that it restructures the overflow periodically, which reduces the amount of time required to build the index (and data).

Programs using BILD$R must be compiled with the $INSERT file OFFCOM from the UFD named SYSCOM (i.e., $INSERT SYSCOM > OFFCOM). Prior to the first call to BILD$R, the user must have assigned PRIMOS file unit numbers to ISUNIT, ISSAM, ISDAM and ISDAT. The template MIDAS file must be open for reading on ISUNIT. The user must also call FILSET to read in the file descriptor, and the flag word in ITEMPS must be set to Ø.

When all entries have been added the user must make a final call to BILD$R for each index with the communication flag set to 2. Otherwise, future additions to the file may overwrite entries added by BILD$R.

If the user is building a MIDAS file with secondary index subfiles using PRIBLD or BILD$R to create the primary index and data, it is not necessary for the user to do anything to set up jarray for the addition of secondary index entries. This is done automatically by the file handler as long as the call to add the primary index/data entry is made immediately prior to the calls to add the secondary index entries for that primary index/data entry.

The calling sequence for BILD$R is:

```
CALL BILD$R (jtemps, keybuf, pbuffr, plenth, jarray,
             index, altrtn)
```

The parameters have the following significance:

Parameter	Meaning
jtemps	A two-word flag and scratch area that permits BILD$R to concurrently build several index subfiles. Jtemps (1) is the communication flag for BILD$R. The remaining words are used by BILD$R and must not be modified by the user.
keybuf	Refer to key in call to ADD1$ (Section 10)
pbuffr	Refer to buffer in call to ADD1$.
plenth	Refer to plenth in call to ADD1$.
jarray	Refer to array in call to ADD1$.
index	Refer to index in call to ADD1$.
altrtn	An alternate return to be taken in the event an error occurs. If altrtn is zero, the program aborts to PRIMOS. The cause of the error will be printed on the user's terminal.

SECTION 8

CONSIDERATIONS FOR SINGLE-USER PROGRAMMING

INTRODUCTION

This section describes a number of subroutines to aid the usage of single-user procedures. Single-user procedures are those MIDAS subroutines that must be used with MIDAS files when no other user is accessing them. Generally these subroutines are PRIBLD, SECBLD and BILD$R, described in Section 7.

OFFCOM

OFFCOM is a $INSERT file that defines 'in a common block' several variables needed by the user for communication with MIDAS. When writing single-user routines that call MIDAS routines, the statement:

```
$INSERT SYSCOM>OFFCOM
```

must be included in each routine that calls one or more MIDAS routines. The FORTRAN $INSERT statement begins in column 1, and must immediately follow the declaration of any COMMON area defined by the user, and must precede any DATA statements. OFFCOM resides in the UFD named SYSCOM and must not be moved from there to the user's UFD. The $INSERT command to the compiler specifies a treename.

MIDAS ROUTINES IN USER MODULE

The user may call any of the routines from the MIDAS file handler without conflicting with the single-user routines. The MIDAS file handler may be used to read from or write to other MIDAS files. For example, one file may be read to create another if some major changes in file format is taking place. However, the single-user routines normally only deal with one MIDAS file at a time.

If BILD$R is being used, and more than one MIDAS file is being accessed, the user must call KX$FCL before and after each call to BILD$R. The call is:

```
CALL KX$FCL
```

OTHER USEFUL MIDAS ROUTINES

FILSET is a routine that reads in the file descriptor subfile (template). The user may find this and some of the other KIDALD routines useful and these routines are described below.

KX$OIX

KX$OIX is a routine to determine the number of entries currently indexed by an index. The calling sequence is:

```
LINDEX = index
CALL KX$OIX (RNUM)
```

RNUM	will contain the number of entries on return from KX$OIX. RNUM is a REAL variable.
LINDEX	is a variable defined in OFFCOM.
index	is the number of the index to be examined.

An invalid index number in LINDEX causes KX$OIX to abort. KX$OIX also reports the contents of the index on the user's terminal.

SYSINI

SYSINI is a routine to close all PRIMOS file units except Unit 6. Calling sequence:

```
CALL SYSINI
```

SYSINI closes all open PRIMOS file units except Unit 6, which is usually the PRIMOS unit open for a command file. SYSINI is a useful routine to call, but at the beginning and the end of a program to ensure that no stray file units remain open from previous procedures and that none are open upon exit from the current procedure.

FILSET

FILSET is a routine to read in the file descriptor subfile. Calling Sequence:

```
ISSAM=funit
CALL FILSET
```

ISSAM is a variable defined in OFFCOM and funit is the PRIMOS file unit assigned to it.

FILSET reads the first 296 words of the file descriptor subfile into INFO in the KIDALB common and otherwise initializes the file. The 296 words are required for all single-user KIDALB routines. Only PRIBLD reads them in independently. If the user wishes to modify these 296 words, they must be written back to the file with a call to KX$BWT.

KX$BWT

KX$BWT is a routine to access the file descriptor subfile.

Calling Sequence:

```
CALL KX$BWT (KEYS)
```

KEYS is an integer variable set to:

:11 - Read Descriptor
:12 - Write Descriptor

KX$BWT opens the file descriptor subfile on ISSAM if necessary. When reading the file descriptor subfile, KX$BWT also makes a cursory check to see if it is valid. FILSET calls KX$BWT.

ERROPN

ERROPN is a routine to open a global logging file.

Calling Sequence:

```
CALL ERROPN (funit)
```

funit is the PRIMOS file unit to be used for the error/log file.

The MIDAS routines FILERR, FILHER and KX$TIM print messages on the user's terminal regarding error or milestone occurrences. These messages can also be written to a file if the user, first, calls ERROPN to open a logging file. The user may also add messages to the logging file by setting up his message in a buffer and then writing it out with a call to the IOCS library routine O$AD07.

The user supplies ERROPN with the file unit for the log file. ERROPN then requests the name of the file to be used. The name may be supplied as a treename.

FILERR

FILERR is a routine to set up and display an error message.

Calling Sequence:

```
CALL FILERR (irout, msg, numb, ialter)
```

irout is the name of the routine calling FILERR. It is a text string with a length of six characters.

msg is the ASCII text to be printed by ERRPR$

numb is the number of characters in MSG

ialter is the user's error return.

FILERR calls the system routine ERRPR$ to print the system error message relating to a nonzero error code returned from one of the system routines. The user may append a message as well.

Routines calling FILERR must be compiled with KPARAM from the UFD named KI/DA as a $INSERT file. KPARAM defines the integer variable CODE which FILERR assumes contains the System Error Code for the ERRPR$ message. To print his own message, the user may set CODE to 37.

The message is displayed on the user's terminal, and if ERROPN has been called, the message is written to the logging file.

If IALTER is zero or not supplied, FILERR closes all open file units and aborts to the monitor.

For the call - CALL FILERR ('KX$BWT', 'BAD FILE', 8,$1000), the message printed by FILERR is for example:

file system error message. BAD FILE (K$BWT)

FILHER

FILHER is a routine to convert and print a MIDAS error code.
Calling Sequence:

CALL FILHER (ierror, ialter)

ierror	is the MIDAS error code
ialter	is the user's error return. If IALTER is 0, FILHER will close all file units and abort to PRIMOS.

FILHER converts a MIDAS error code to ASCII and calls FILERR to print the result. If the error code is greater than 12, FILHER aborts to PRIMOS.

KX$TIM

KX$TIM is a routine to print a milestone/time record.
Calling Sequence:

CALL KX$TIM (lnum,timfil,text,txlent)

lnum	is a long integer which represents, for example, a record count. If lnum is 0, a header record also is printed.
timfil	is the PRIMOS file unit to which the message is to be written. If timfil is 0, the message only appears on the user's terminal.
text	is an additional message to be written out before the time record. If txlent is 0, text may be 0 also.
txlent	is the length of text in words. If txlent is zero, no message corresponding to text will be written.

KX$TIM prints a time difference record with additional user information. For example the message:

```
END OF RUN

      103    05-23-77    12:40:56    0.882    0.354    1.236    0.074
```

was printed by KX$TIM.

The example above could have been printed by the statement:

```
CALL KX$TIM(000103,4,'END OF RUN',5)
```

The message would appear on the user's terminal and would also be written out to the file open on file unit 4.

The following call:

```
CALL KX$TIM (000000,0,0,0)
```

results in the generation of a header as follows:

COUNT	DATE	TIME	CPU MIN	DISK MIN	TOTAL TM	DIFF
0	05-23-77	12:34:52	0.000	0.000	0.000	0.000

The columns in the message are as follows:

COUNT	number of entries processed (up to 2**31-1)
DATE AND TIME	date and time of entry (Time hrs., mins.,secs.)
CPU MIN	minutes of CPU time used since start of run
DISK MIN	minutes of disk I/O time (including paging) from start
TOTAL TM	sum of disk and CPU time since start
DIFF	increment in total time since last call to KX$TIM

CAUTION

If calling any other routines in that reside in the library KIDALB, remember that most of these routines make assumptions about the contents of COMMON which the user may not have anticipated.

SECTION 9

DATA ACCESS SUBROUTINES

The subroutines described in this section make it possible to access a MIDAS file for read, write or update functions in a multi-user environment. The MIDAS file handler takes care of file integrity so that two or more users may not update or add to the same record at the same time. In order to achieve multi-user access, two system wide conventions are required. These are:

1. The READ/WRITE lock, RWLOCK, must be set to 3. (Note: on some systems, the individual file protection attributes for MIDAS files may be set at 2). This setting allows multiple readers, and one writer, to access the MIDAS file. The system (or file) must not be set up for access by multiple writers.

2. Each user of the multi-user subroutines must open MIDAS files for reading only using the PRIMOS file system routine SRCH$$. This is true even if the user plans to update or add records to the file. MIDAS opens the file for writing when necessary and returns the file to read-only access upon closing it.

Data access subroutines are called directly by FORTRAN and PMA programs, and they are part of the run-time support package for indexed file processing with COBOL and RPG. Data access routines provided by MIDAS are:

Routine	Function
ADD1$	Adds a data entry to a MIDAS file.
DELET$	Deletes a data entry from a MIDAS file.
FIND$	Finds a data entry on a MIDAS file.
LOCK$	Finds a data entry on a MIDAS file and locks it for safe updating in the multi-terminal environment.
NEXT$	Finds the next data entry in a MIDAS file.
UPDAT$	Updates a data entry on a MIDAS file.

With respect to the data transferred, the primary key of the entry accessed may be included or excluded from the user's entry. The desired option is selected by setting a bit in the calling sequence flag word.

(flags). This feature is useful when accessing entries through the secondary indexes when the data is being traversed sequentially.

Unless otherwise specified, all variables and arrays used as parameters for calls to the MIDAS file handler data access routines must be defined in INTEGER*2.

CALLING SEQUENCES

General Format

The six MIDAS data access subroutines have essentially the same FORTRAN calling sequence; therefore, the parameter list is defined only once. Individual variations in parameter values are discussed with the appropriate data access subroutine. The calling sequence is:

```
CALL Rtn (funit, buffer, key, array, flags, altrtn,
          index, file-no, plenth, keylnt
```

Rtn is the name of one of six MIDAS data access routines: FIND$, LOCK$, UPDAT$, ADD1$, DELET$, NEXT$. Further on in this section each routine is described individually. Associated with each description is an example of the use of the routine. The examples supply a consistent picture of how to use MIDAS in a simple situation. In the examples the following variables are used:

FUNIT PRIMOS file unit on which the MIDAS file is open.

BUFFER The data record buffer. Dimensioned to 80 for demonstration purposes.

PKEY Buffer for the primary log. Dimensioned to 6 for demonstration purposes. Equivalenced to BUFFER (,).

SKEY-2 Buffer for a secondary key to be used with secondary index 2. Dimensioned to 4 for demonstration purposes.

RKEY-3 Variable holding key value for secondary index 3. The key is REAL for demonstration purposes.

FLAGS The MIDAS file handler flag word

ARRAY The MIDAS file handler communications array

Description of Parameters

Table 9-1 defines the purpose and content of the parameters in a call to a MIDAS data access subroutine.

Table 9-1. Parameters for Data Access Subroutines

Parameter	Meaning
funit	is a PRIMOS File Unit on which the MIDAS file is open.
buffer	is a one-dimensional array into which, or from which data is transferred.
key	is a numeric variable or a one-dimensional integer array containing the key field value to be used in locating the data entry.
array	is a 14-word integer array used for file handler and user communications. Format and values contained in array are discussed in the following paragraphs.
flags	is a integer variable used to select options for the call. Values and format of flags are given in the following paragraphs.
altrtn	is an alternate return to be taken by the routine if an error occurs.
index	is a short integer variable that identifies the access method to be used (Keyed Index or Direct Access) and/or selects the index to be used. Values for index are given in the following paragraphs.
file-no	a short integer variable that, at present, must be set to 0 or -1. It is provided for subsequent index retention and sharing on Prime 400 computer systems. Note, past versions of MIDAS included a functionality for this argument. This functionality is still available as previously described but it is of little use to most users, unless they are operating under an old software revision.
plenth	is a short integer variable which contains the length of the data to be transferred. May be zero if the full data entry is to be transferred. This parameter is not required or used for calls to DELET$.
keylnt	is a short integer variable containing the key length to be used. It is expressed in words or bits (bytes) as determined by flags. Keylnt may be zero if the full key is to be used. This parameter is not required or used for calls to UPDAT$, DELETE$, ADD1$ or LOCK$.

The following paragraphs amplify some of the more complex parameters in the calling sequence for a MIDAS data access routine.

Format of Index

The index parameter specifies whether the file is to be accessed through a keyed-index or direct access method. For the keyed-index method, the index parameter also specifies the index-value that is to be used. The format for the index parameter is:

	Direct Access	Indexed Access
bits 1-8	all 1's	all 0's
bits 9-16	all 1's	Index number (primary index is 0)

Format of Flags

The parameter flags in a data access calling sequence provides user options within the routine that is being called. At present, up to ten bits of the flags parameter are used. The remaining bits are reserved for use by the MIDAS file system.

The $INSERT file PARM.K in the UFD named SYSCOM contains mnemonics for the flag bits for use in the user's MIDAS applications. To compile with the $INSERT file include the statement: $INSERT SYSCOM > PARM.K. Accordingly, the PARM.K mnemonics are included in the descriptions below. Parameters in the $INSERT file may also be efficiently summed to generate the appropriate constant without generating a number of useless constants.

For example:

FL$USE + FL$RET + FL$PLW complies as :144000.

The bit settings in flags are summarized in Table 9-2.

Table 9-2. Bit Settings for Flags Parameter

Bit	Mnemonic	Setting	Meaning
1	FL$USE	SET	Use contents of array
		RESET	Ignore contents of array
2	FL$RET	SET	Return contents of array
		RESET	Return completion code only.
3	FL$KEY	SET	Include primary key in the data buffer, "Buffer".
		RESET	Do not include primary key in Buffer.
		For NEXT$ and FIND$ only:	
4	FL$BIT	SET	Key size is in bits or bytes
		RESET	Key size is in words
5	FL$PLW	SET	Get next entry, regardless.
		RESET	Get next entry only if key agrees with search key.
6	FL$UKY	SET	Update key field with key from data base.
		RESET	Do not change original contents of key field.
7	FL$SEC	SET	Return secondary data from secondary index.
		RESET	Access primary data record.
		For UPDAT$ ONLY:	
8	FL$ULK	SET	Update the setting of the data.
		RESET	Update data entry.
		For FIND$, LOCK$, and NEXT$:	
9	FL$FST	SET	Position to first index entry
		RESET	Use user-supplied search key.
		For FIND$ AND NEXT$:	
10	FL$NXT	SET	Get next record greater than supplied key.
		RESET	Get record specified by supplied key.
11-16		MBZ	Must be zero.

Use of FL$KEY

In the physical data subfile, MIDAS stores a data record that has the format shown in Figure 9-1A. Since MIDAS keeps a copy of the primary key (PKEY), it is unnecessary for the user to maintain a copy also, as shown in Figure 9-1B or C. Accordingly, MIDAS allows the user to consider his data records as shown in Figure 9-1D, where the value of PKEY is MIDAS', i.e., a user with a type B record can retrieve it with either one or two copies as in Figure 9-1E. This is even more obvious for users with records of this type shown in Figure 9-1F.

Users probably like to be consistent. For those users who like to consider the primary key as part of their data record, are willing to have it as the first field in the data record, and do not want two copies of the primary key in the data subfile, the following provision exists:

1. Put the primary key (PKEY) in the users copy of the data buffer.

2. Set FL$KEY.

3. For a call to ADD1$, MIDAS writes only one copy of key.

Format of Array

General Information: The parameter array is an array of either one word or 14 words that is used by the MIDAS file handler to return a completion code. If Bits 1 and 2 (F$USE and F$RET) of the parameter flags are RESET, then array may have a length of one word (i.e., two bytes), otherwise, array must have a length of 14 words (28 bytes). For most purposes, an array size of 14 words is recommended.

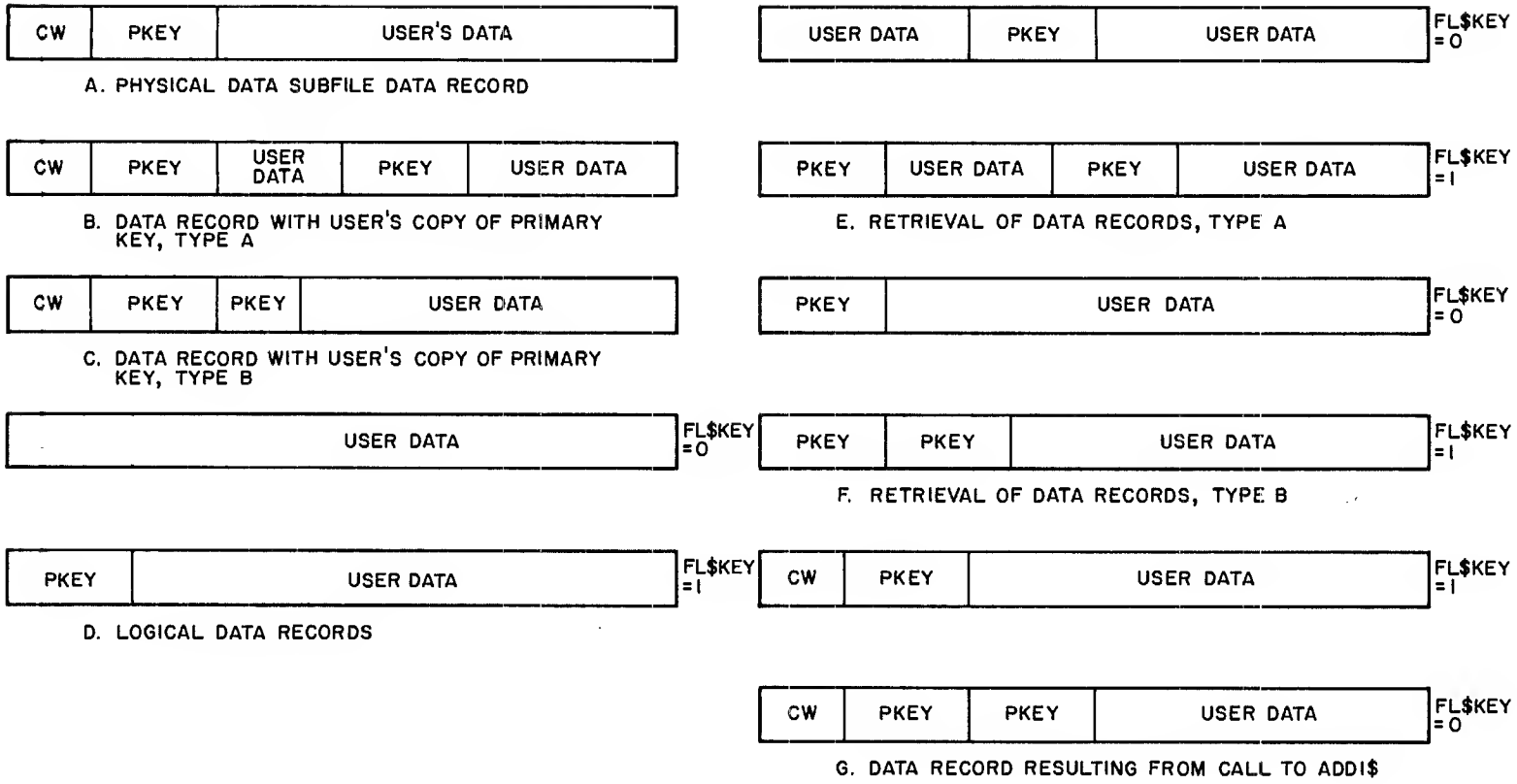


Figure 9-1. Duplicate Key Examples

Indexed Access: The general format of array for indexed access is as follows:

Word	Bit	Setting	Contents and/or meaning
1	-	0 or 1	Contents of array are valid
		-1	Contents of array are not valid. For condition codes, see Appendix A.
Words 2-12	-	-	Used by the file handler. Not to be referenced or modified by user.
Word 13	1-4	0	Flags for file handler (DE$LOK)
	5		SET: Record locked CLEAR: Record not locked
	6-16		Flags for file handler
Word 14	-	-	Length of data entry (exclusive of length of primary key).

If a user sets FL$USE in flags, the only valid modification of Array is Word 1 to -1. This overrides FL$USE set to 1 and allows a user to make the first call in a NEXT$ loop with the same flags value as is used in subsequent calls in the loop. Any other tampering with the contents of Array may cause trouble.

A user wishing to know if another user is in the process of updating the data record may examine the bit 5 flag to find out.

Direct Access: The following list shows the format of Array when the method is Direct Access:

Word	Contents and/or meaning
1	Array state or condition code word.
2	Entry length (key length + user's data +2.)
3-4	A single-precision floating point entry number
5-14	Same use and format as for the corresponding bits in indexed access.

ADD1$

The ADD1$ subroutine adds an entry to a specified data or index subfile. ADD1$ has no control over access methods. The method required by ADD1$ is established when CREATK is used to create the template file. The user must add records according to this pre-determined file definition.

Synonym entries (i.e., duplicate keys) may only be added through secondary index access, and they are only allowed if permitted by the user's file structure. Other special consideration for secondary indexes are discussed in the following paragraphs. A copy of the array returned by a call to ADD1$ may not be used by a subsequent call to NEXT$, but it may be used for re-reading the entry or for adding additional secondary index entries.

ADD1$ can be used to add a secondary index to an existing data entry in the MIDAS file. When adding secondary index entries, the user must supply the following information:

1. Secondary index number
2. Secondary key value
3. Primary key value
4. Secondary data (if any)

The secondary index number is supplied as a one word integer in the argument index to ADD1$. The secondary key value is supplied in the argument key.

The primary key value is supplied as the first item in the argument array buffer. If there is secondary data, the data follows the primary key in buffer.

The user may either request the MIDAS file handler to locate the primary data entry (FL$USE reset), or may use an existing, valid copy of array (FL$USE set to 1). In particular, array (2-14) must be correct for the entry to be added. An array returned by a previous call to ADD1$ for the same primary key is satisfactory.

If plenth is 0 (undefined), the parameter buffer is assumed to contain the primary key and any secondary data specified by the file description. If plenth is not 0, the length supplied must include the primary key as well as the length of any user data.

For both the keyed index and direct access methods, data entries can be added by the primary key only.

When ADD1$ is adding information to a file, the file is locked to other user additions until operations are complete.

The parameter settings for each option are discussed in the following paragraphs.

Access Methods

ADD1$ can operate in either of two access methods, Keyed Index or Direct Access.

Data entries can be added to a direct access file (MIDAS' DAM ≠ PRIMOS DAM) by direct access only. Entries to a non-direct access file must be added through the primary index.

The direct access mechanism is specified by setting the index parameter to the value -1. Any other legal value indicates that the keyed index access method is required.

Keyed Index Method: The keyed index method enables data entries and corresponding primary index entries to be added to the primary index. Secondary index entries may be added to the existing data entry for one of 19 secondary indexes. The index structure to be used is specified in parameter index. Possible values are:

Value	Meaning
-1	See Direct Access Method
0	Primary Index
1 to 19	Specified Secondary Index

For index values of 0, the data entry information must be supplied in buffer. For index values of 1-19, the primary key must be at least specified in first item in the array buffer, and the corresponding full secondary key must be specified in key.

When adding data entries or index entries to a MIDAS file, full sized keys (i.e., not partial) must always be supplied in parameter key. The parameter keyInt (key length) may be omitted. Therefore, Bit 4 (FL$BIT) in flags is not applicable, in this case.

Direct Access Method: The direct access method supplements the keyed index method of retrieval and addition of entries via ADD1$. It is applicable to MIDAS files with CREATK templates that specify the direct access mechanism (i.e., for the direct access mechanism to apply the data entry subfile of a MIDAS file must be pre-allocated on disk). On such files, addition of data and primary index entries is illegal.

To ADD a record through direct access the user must supply a full primary key, a floating point data entry number and the full data entry size specified in words. The data entry number need not be related to the primary key in any way.

The data entry number is passed to the file handler in words 3 and 4 of array.

The data entry size is the length of the data record as stored on disk (i.e. 2 + Key length in words + data length in words). It is passed to the file handler as word 2 of array.

Hence the first 4 words of array contain:

Word 1	Condition code (0 or 1)
Word 2	Entry size
Word 3-4	"Entry number" in REAL format.

Index must be set to -1. Key must contain the desired full Primary Key value.

If an entry is already found for the Data Entry Number, MIDAS resolves the conflict by placing the second data entry in an overflow area.

A primary index entry is created and added to the file, making it possible to use NEXT$ on direct access MIDAS files through Primary Index access.

Adding Information

ADD1$ adds data to a MIDAS file in one of two ways:

1. Add a new data entry with associated primary key.
2. Add a new secondary index with user data, if any, to an existing secondary index.

There are no user options to distinguish adding of primary index entries from the adding of data.

Use of Buffer Parameter

The information required in buffer differs, depending upon whether a primary index entry/data entry is being added or whether a secondary index entry is being added to an existing data entry.

Primary Index Entry/Data Entry: buffer must contain the full sized data entry information. For fixed data length MIDAS files, the parameter plenth must be set to the default value of 0. Variable length files require a plenth that is the length of the data to be added.

When writing the data to the data entry on file, the primary key value is added from parameter key and will procede the data entry in the data entry written to disk.

Secondary Index Entry: buffer must contain the full sized primary key value. Furthermore, if supplementary information is required to be stored in the secondary index entry, the supplementary data must follow the primary key value. The parimary key value is required to locate the data entry to which the specified secondary index is applicable.

Use of Plenth Parameter

Primary Index Entry/Data Entry: The plenth parameter must be set according to the type of data entry:

Data Entry	Plenth
Fixed-length	0
Variable - length	Length of data

Secondary Index Entry: If plenth=0, ADD1$ adds the entire contents of buffer - the primary key plus secondary data as specified in the file description.

If plenth is set to the length of the primary key plus part of the secondary data, only that part will be added. (The entry will be zero-filled if necessary.)

Notes

1. To add a secondary index entry, (FL$USE) of flags must be set. There must have been an immediate prior call to FIND$ on the primary key, specifying FL$RET in flags, i.e., return array, and FL$KEY must be set to include the full primary key in buffer.

2. To add a primary index entry, FL$USE of flags will be ignored: that is, ADD1$ locates the point in the index structure to add the data entry.

3. ADD1$ returns an array that is suitable only for index entries.

Use of Array Parameter

The array parameter defines a scratch area used by the MIDAS file handler. The first word of array is always used to return completion codes or error codes to the caller. Possible values are:

Value	Meaning
0	Successful completion
1	Successful completion. There is a synonym (secondary indexes only)
12	This is a synonym and synonyms are not allowed (primary indexes and specified secondary indices specified at file creation time - see Appendix A).
none of above	Refer to the Error Message summary in Appendix A.

By setting FL$RET in flags, the option may be called that returns the full scratch values for the current access. The values contained in this array are pointers to the last level index block and data entry.

Table 9-3. Summary of Parameter Values for ADD1$

Parameter	Options (if any)	Meaning
funit		File unit on which the specified file's segment directory is open.
buffer		Buffer in which data to be added is stored. Buffer must include a Primary Key if a secondary index add is specified and must include any data needed to add to the secondary indexes.
key		Full primary or secondary key appropriate to the index.
array		Full primary or secondary key appropriate to the index.
flags		Options that specify use of various parameters. The options for ADD1$ are:
	FL$USE	Use contents of array Not required for adding a primary index and data.
	FL$RET	Return contents of array. Only applicable for a primary or direct access add.
altrtn		An alternate return to be taken if an error occurs.
index		0 = Primary Index; Nonzero = Secondary Index; - 1 = Direct Access.
plenth		Length of data (see text).

When using ADD1$, the only time the returned parameter array is useful is for subsequent calls to add a secondary index. After such a call to add a secondary index entry, the contents returned in array are still useful for adding a further secondary index entry.

By setting FL$USE in flags the option may be called that short circuits the index search mechanism provided the array contents are valid. If the user is adding a secondary index, then this option may be used for adding secondary index entries to an existing data entry which has already been located.

When using the direct access facility and a primary index entry/data entry is being added, array contains the user supplied Data Entry Number and entry size. In this case, FL$USE in flags may be set; it is used anyway. Also, with direct access, the array that is returned is useful for the addition of secondary index additions. However, it is meaningless for other MIDAS operations.

Using ADD1$ - Examples

The following code adds a data record to a MIDAS file and adds a secondary index entry for the same data record:

```
.
.
.
INTEGER * 2 FLAGS, BUFFER (80), PKEY (6), FUNIT, ARRAY (14),
+ SKEY 2 (4)

EQUIVALENCE (BUFFER (1), PKEY (,))

.
.
.
FLAGS = FL$KEY + FL$RET /* RETURN ARRAY, KEY IN BUFFER
CALL ADD1$ (FUNIT, BUFFER, PKEY, ARRAY, FLAGS, $9000,0,0,0)
FLAGS = FL$USE
CALL ADD1$ (FUNIT, BUFFER, SKEY-2, ARRAY, FLAGS, $9100,2,0,0)
.
.
.
```

The following code adds a secondary index entry to a MIDAS file data entry that has not been recently accessed:

```
.
.
.
INTEGER * 2 FLAGS, BUFFER (80), PKEY (6), FUNIT, ARRAY (14)
REAL PKEY-3

EQUIVALENCE (BUFFER (1), PKEY (,))
.
.
.
FLAGS = Ø
CALL ADD1$ (FUNIT, PKEY, PKEY-3, ARRAY, FLAGS, $9200,3,0,0)
.
.
.
```

DELET$

DELET$ locates a data entry by either the keyed index or direct access method and deletes the data entry and/or primary index entry. The secondary index entry is deleted if a secondary index is specified. DELET$ ignores any data entry lockouts that may be have been applied.

Where synonyms are present in the file (secondary indexes only), DELET$ locates the first only (oldest synonym). NEXT$ must be used to position DELET$ in this case.

Calling Sequence

DELET$ is called using the standard call described in this section. The parameter values involved in the call to DELET$ are used to specify the options available in the following areas:

1. Access Methods
2. Use of array

The parameter settings for each option are discussed in the following paragraphs.

Access Methods

Before deletion, the data entry may first be located by FIND$, if there are no synonym (duplicate) keys. NEXT$ must be used if there are synonym (duplicate) keys. The basic access method for DELET$ is keyed index. The direct access method is applicable if Direct Access was specified when the template was created by CREATK.

The direct access mechanism is specified by setting the index parameter to the value -1. Any positive value indicates that the keyed index method is required.

Keyed Index Method

Choice of Index: The keyed index method enables data entries to be located by searches on the primary or one of the secondary index structures. If index is 0 or -1, it means Delete data; if index is greater that 0, it means: delete the specified secondary index entry. The index structure to be used is specified in the index parameter.

Possible values are:

Value	Meaning
-1	See Direct Access method
0	Primary Index
1 to 19	Specified secondary index

If direct access or a primary index are specified, the data entry and the primary index entry are deleted and all related secondary index entries are rendered void. During the next run of REMAKE, these voided secondary index entries are deleted. This accounts for the difference in secondary index entry counts before and after a REMAKE.

If the secondary index is specified, only the specified secondary index entry is deleted.

Full or Partial Key Values: When using the keyed index method, the user must supply a full key (primary or secondary) to DELET$. Partial keys are illegal because positive identification of the entry to be deleted is required. The user supplies a key value in parameter key.

Direct Access Method

The direct access method is applicable to MIDAS files with templates that were created specifying direct access. For the direct access mechanism to apply, the records of the data entry subfile of a MIDAS file must be preallocated on disk. On any such MIDAS file, all keyed index operations (i.e., full key searches on secondary indexes for deletion of secondary index entries) still apply.

To delete a record through direct access the user must supply a full primary key, a floating point data entry number and the full data entry size specified in words. The data entry number need not be related to the primary key in any way.

The data entry number is passed to the file handler in words 3 and 4 of array.

The data entry size is the length of the data record as stored on disk (i.e., 2 words + Key length in words + data length in words). It is passed to the file handler as word 2 of array.

Hence the first 4 words of array contain:

Word 1	Condition code. (0 or 1)
Word 2	Entry size
Word 3-4	'Entry number' in REAL format.

Other parameters are:

index	must be set to -1
key	must contain the desired full Primary Key value.
flags	FL$USE may be zero; array is used anyway.

If two or more primary key values are found at the same data entry number, MIDAS resolves this conflict by the normal index search. Consequently, a slight increase in search time results.

Use of Array

The array is a scratch area used by the file management system. Consult Table 9-4 for detailed contents. The first word of array is used to return completion codes or error codes to the caller. Possible values are:

Value	Meaning
0	Successful deletion
1	Successful deletion but there may be further synonyms on the file
7	Entry not found
other	Error code (refer to Appendix A)

When using the direct access facility, array is used to contain the user supplied Data Entry Number. With direct access, it is meaningless to set FL$RET since the returned array cannot be used.

By setting FL$USE in flags, the user may supply valid array contents. These must have been generated by a previous call to FIND$, NEXT$, or LOCK$ that located the data entry on the same index whose index entry (and data entry for primary indexes) is required to be deleted.

Summary of Parameters for DELET$

Table 9-4 summarizes possible values for parameters and options specified in a call to DELET$.

Table 9-4. Summary of Parameter Values for DELET$

Parameter	Options (if any)	Meaning
funit		PRIMOS file unit on which this files segment directory is opened. This specifies the MIDAS file being used.
buffer		Not applicable
key		Full primary or secondary key to be used to identify the data entry.
array		14-word block used for file handler/user communication and for direct access only. For direct access, array is used to input the user-supplied data entry number and entrysize.
flags		Options that specify use of array parameter. The options for DELET$ are described in the following table entries.
	FL$USE	For direct access: May be =0; array is used anyway. For keyed index: 1 = use contents of array. Array must be set up from a previous successful call to one of the file system routines (i.e., have a condition code of 0 or 1, successful retrieval). By supplying a valid copy of array, the same data that was retrieved from a previous call (FIND$, NEXT$, etc.) will be deleted. All other bits in flags are ignored except FL$RET.
altrtn		An alternate return to be taken if an error occurs.

Parameter	Options (if any)	Meaning
index		Index to be used - Its values are: Value — Meaning 0 — Primary 1 — Secondary 2 — Data ... -1 — Direct Access Values of 0 or -1 cause both the data entry and primary index entry to be deleted. Any other value causes the appropriate secondary index entry to be deleted.
file-no.		Ø or -1
plenth		Not used
keylnt		Not used

Using DELET$

Following is an example of the use of DELET$ to delete a data record and all secondary index entries associated with it.

```
.
.
.
INTEGER *2 FLAGS, BUFFER (80), PKEY (6), FUNIT, ARRAY (14)
EQUIVALENCE (BUFFER (1), PKEY (1))

.
.
.
FLAGS = FL$KEY
CALL DELET$ (FUNIT, BUFFER (80), PKEY (6), ARRAY, FLAGS, $9000,0,0)

.
.
.
```

To delete a secondary index entry when there are duplicate occurrences of the key, as in:

```
.
.
.
INTEGER *2 FLAGS, BUFFER (80), PKEY (6), FUNIT, ARRAY (14),
PTEST (6) EQUIVALENCE (BUFFER (1), PKEY (1))
REAL RKEY 3

.
.
.
FLAGS = FL$RET + FL$USE + FL$KEY
ARRAY (1) = -1
100 CALL NEXT$ (FUNIT, PTEST, RKEY-3, ARRAY, FLAGS, $9000,3,0,6,0)
DO 200 I = 1,6
IF (PTEST (I) .NE. PKEY (I) GO TO 100

200 CONTINUE
FLAGS = FL$USE
CALL DELET$ (FUNIT, PTEST, RKEY-3, ARRAY, FLAGS, $9100,3,0)

.
.
.
```

FIND$

The MIDAS routine FIND$ locates a data entry by either the keyed index or direct index method and reads the specified amount of data into the location specified by the buffer parameter. Parameters specified by FIND$ are listed in Table 9-5.

FIND$ ignores any data entry lockouts that may have been applied. Where synonyms are present on the file (secondary indexes only), FIND$ locates the first synonym only (oldest synonym).

Calling Sequence

FIND$ is called using the standard call described previously. The parameter values in the CALL to FIND$ are used to specify the options available in the following areas:

1. Access Methods
2. Information retrieved
3. Use of user-supplied array

The parameter values for each option are discussed in the following paragraphs.

Access Methods Supported by FIND$

There are two access methods supported by FIND$. These are:

1. Keyed Index
2. Direct Access

The most commonly used access method is keyed index. The direct access method is potentially faster and is applicable if a full primary key value is supplied and Direct Access was specified when the file template was created by CREATK. The direct access mechanism is specified by setting the index parameter to the value -1. Any other allowable value indicates that the keyed index method is required.

Keyed Index Access Method

Choice of Index: The keyed index method enables data entries to be retrieved by searches on either the primary or one of the secondary index structures. The index structure to be used is specified in parameter index as follows:

index	Meaning
-1	Refer to Direct Access Method
0	Primary Index
1 to 19	Specified secondary index

The user must supply a primary key value if the primary index is specified, or supply a first secondary key if the first secondary index is specified, etc.

Full or Partial Key Values: When using the keyed index method, full keys or partial keys (both primary and secondary) may be supplied to FIND$ for retrieval of the required data entry.

Partial keys are truncated full keys (for example, the first four characters of an eight-character full key). When partial keys are supplied, FIND$ returns the first data entry that has a full key that begins with the partial key.

Use of partial keys is specified in the parameter keylnt which takes the following values:

Keylnt	Meaning
0	Use full sized key
non-zero	Size of partial key in bits, bytes or words

The value of the partial key size is either in bits, bytes or words according to the value of FL$BIT in parameter flags, namely: RESET = partial key size in words or; SET = partial key size in bits or bytes (always supplied in key). Key values are specified in parameter keys.

Direct Access

The direct access mechanism provides an alternative, potentially faster, retrieval mechanism for those retrievals that are based upon a full primary key value being supplied. For the direct access mechanism to be effective, the data entry subfile of a MIDAS file must be pre-allocated on disk.

On any direct access MIDAS file, all keyed index operations i.e., full or partial key searches on primary or secondary indexes function as previously described.

The user must supply a Data Entry Number previously assigned to the data entry.

To find a record through direct access the user must supply a full primary key, a floating point data entry number and the full data entry size specified in words. The data entry number need not be related to the primary key in any way.

The data entry number is passed to the file handler in word 3 and 4 of array.

The data entry size is the length of the data record as stored on disk (i.e., 2 + Key length in words + data length in words). It is passed to the file handler as word 2 of array.

Hence the first 4 words of array contain:

Word 1	Condition Code (0 or 1)
Word 2	Entry size
Word 3-4	'Entry number' in REAL format

The index parameter must be set to -1. Key must contain the desired full primary key value. KeyInt must be set to the default value of zero (i.e., it indicates a full key). Flag FL$USE may be zero; the array will be used anyway.

If two or more primary key values are found at the same Data Entry Number, MIDAS resolves the conflict by the normal index search method. This results in a slight increase in retrieval speed.

Information Retrieved (All Methods)

The user options available in FIND$ enable information to be returned from:

- The data entry
- The primary key value for the data entry
- The secondary index entry and any user data stored with the index entry.
- The full key value.

Use of Buffer: After a successful call to FIND$, MIDAS returns the information from the data entry corresponding to the key supplied or, alternatively, information from the last level secondary index entry in buffer. This latter option is available if a secondary key search is made.

FL$SEC of flags must be SET if it is required to return information from the index entry rather than the data entry.

The user must provide a buffer sufficiently long to contain all the return information, the data entry, and the full primary key if specified by FL$KEY.

If the return of all information is required, plenth may be zero. Otherwise, plenth specifies the total number of words required to be returned, including the full primary key if specified by FL$KEY.

The facility to return the full primary key can be most useful to the user under the following sample conditions:

1. The data entry has been found by supplying a secondary key and it is necessary to know the value of the corresponding primary key.

2. The data entry has been found by supplying a partial primary key and it is necessary to identify the exact data entry accessed by the corresponding full primary key value.

3. The data entry has been found by supplying a partial secondary key, but array indicates that further synonyms exist on the file. The identify of this first synonym can be established by reference to the full primary key returned, which is always unique.

Use of Key: This key parameter provides the information by which FIND$ locates the data entry in the file.

When a partial key is supplied, the data entry retrieved will be the first whose appropriate key gives a match over the partial key length. Sometimes, it is necessary to identify the data entry retrieved by a partial key by having the full secondary or primary key returned also. FL$UKY in flags must be set if this option is required. On return, key contains the full key corresponding to the partial key originally supplied. The user must ensure that the key array is large enough to contain the full key if FL$KEY is set.

Use of Array: The user-supplied array is a scratch area used by the MIDAS file management system.

The first word of array is always used to return completion codes or error codes to the caller. These may be:

Code	Meaning
0	Successful retrieval
1	Successful retrieval but there may be further synonyms on the file
7	Entry not found
other	See Appendix A

By setting FL$RET in flags, the full contents of array are returned for the current access. A user may have the full array returned in order to call other MIDAS routines (such as LOCK$, DELET$, etc.) for the same data entry, perhaps with different optional parameters. The values returned in array allow the file handler to short-circuit the index search mechanism.

When using the direct access facility, array contains the user-supplied data entry number. Also, with Direct Access, the returned array can only be used for addition of secondary index entries or to re-access the same record.

Summary of Parameters for FIND$

Table 9-5 lists the parameters specified in a call to FIND$ along with a brief explanation of their meaning. Further information is given in the general discussion of parameters at the beginning of this section.

Table 9-5. Summary of Parameter Values for FIND$

Parameter	Options (if any)	Meaning
funit		PRIMOS file unit number on which the specified files segment directory is open. This specifies the MIDAS file being used.
buffer		Buffer into which retrieved data is stored.
key		Full or partial primary or secondary key is used to identify the data entry.
array		14-word block used for file handler/ user communication and for direct access only to input the user supplied data entry number and entry size.
flags		Options that specify use of various parameters. The options for FIND$ are described in the following table entries:
	FL$USE	For direct access may be =0 (i.e., RESET); (array is used regardless for direct access). For keyed index, SET bit 1 to use contents of array. Array must be set up from a previous call to one of the MIDAS file system routines and have a condition code of 0 indicating successful retrieval. (A condition code of 1 indicates successful retrieval, but there may be further synonyms.) By supplying an array, the data retrieved from a previous call (FIND$, NEXT$, etc.) is obtained again, but array must be correctly set up. If the index parameter indicates direct access, array must be set up with entry size and entry number.
	FL$BIT	Key size to use is specified in bits or bytes.
	FL$SEC	Return user data from secondary index rather than primary data entry. This is not applicable if primary index access or direct access has been specified.

	FL$KEY	Include primary key in data returned to caller (unless specified, it is not returned). Plenth must be specified with the primary key value in mind.
	FL$UKY	Update keyfield with actual key from data base. This option is useful if FIND$ is being used with partial keys.
	FL$RET	Return full array; otherwise, only return completion code.
altrtn		An alternate return to be taken in the event of an error.
index		Index to be used: 0 - Primary 1 to 19 - Secondary -1 - Direct Access
file no.		0 or -1
plenth		Length of input data. Ø means full amount of data entry plus full primary key if specified by FL$KEY.
keylnt		Length of key to be used. Zero means full size as in data base; nonzero means partial key. A direct access must use the default value of 0, i.e., it must contain the full key.

Using FIND$

The following example retrieves a data record from a MIDAS file via the primary index:

```
.
.
.
INTEGER *2 FLAGS, BUFFER (80), PKEY (6), FUNIT, ARRAY (14)
EQUIVALENCE (BUFFER (1), PKEY (1))

.
.
.
FLAGS = FL$KEY + FL$RET 1* SET UP TO RETURN ARRAY AND PKEY
CALL FIND$ ( FUNIT, BUFFER, PKEY, ARRAY, FLAGS, $9000,0,0,0,0)
```

This example retrieves a data record from a MIDAS file via a REAL secondary key:

```
.
.
.
INTEGER *2 FLAGS, BUFFER (80), FUNIT, ARRAY (14), PKEY (6)
REAL RKEY-3
EQUIVALENCE (BUFFER (1), PKEY (1))

.
.
.
FLAGS = FL$KEY + FL$RET
CALL FIND$ (FUNIT, BUFFER, RKEY-3, ARRAY, FLAGS, $9000,3,0,0,0)

.
.
.
```

FIND$ - EXAMPLE

This example retrieves secondary data from a KI/DA file (index 2)

```
.
.
.
INTEGER *2 FLAGS, BUFFER (80), FUNIT, ARRAY (14), PKEY (6),
SKEY 2 (4)
EQUIVALENCE (BUFFER (1), PKEY (1))

.
.
.
FLAGS = FL$KEY + FL$SEC + FL$RET
CALL FIND$ (FUNIT, BUFFER, SKEY 4, ARRAY, FLAGS, $9000,2,0,0,0)

.
.
.
```

LOCK$

LOCK$ locates a data entry by either the keyed index (any index) or direct access method. If the data entry is not locked LOCK$ reads into buffer the specified amount of data. Afterwards, LOCK$ locks the data entry.

The action of LOCK$ is similar to FIND$, with some differences caused by inclusion of the data entry lock feature. These differences are discussed in the following paragraphs.

Data Entry Lockout

Data entry lockout protects a data entry on a MIDAS file between retrieval and subsequent update. This protection prevents other users (also using LOCK$ rather than FIND$) from retrieving the same data entry and updating it. If this happened, the result of the first update would be lost. LOCK$ protects the data entry from retrieval or update by any of the possible primary or secondary index values.

LOCK$ first tests that the data entry is not already locked before retrieval. After successful retrieval, the data entry is locked. UPDAT$ must be the next routine called for the file retrieved, and there must be no other access to the locked record by the locking user until then. FIND$, NEXT$, AND DELET$ may not be used to access the record by other users while it is locked.

Calling Sequence

LOCK$ may be called specifying any index, and it prevents update by other users asking for retrieval, while LOCK$ is operating.

Since UPDAT$ must be the next routine called after LOCK$, and UPDAT$ must have array supplied, FL$RET in flags must be set to cause return of the array contents. Array is valid whether LOCK$ used the keyed index or direct access method specified in the index parameter of the call.

The data entry being locked must be identified fully. The user must supply a full key value in the key parameter and the key must be appropriate to the index that is used. KeyInt is ignored.

Use of Array

Array can be supplied on input (i.e., FL$USE set in flags). This is appropriate when it is required to lock a data entry after a call to FIND$ or NEXT$ for the same data entry. Using this technique, the data entry can be identified before the lock is applied. Index traversals can be short circuited when calling LOCK$.

The first word of the returned array always returns completion codes or error codes to the caller. The meanings of the contents of this word are:

Value	Meaning
0	Successful retrieval
1	Successful retrieval but there may be further synonyms on the file.
7	Entry not found
10	Entry found, but already locked.
Other	See Appendix A.

Summary of Parameters for LOCK$

Table 9-6 lists and briefly describes the parameters and options for LOCK$.

Notes

1. If LOCK$ is being used to locate the record, a full key for the selected index (or direct access) must be supplied.

2. If the record has been found by FIND$ or NEXT$ and FL$USE is set in flags, no key is required or used as the data entry has already been found.

Table 9-6. Summary of Parameter Values for LOCK$

Parameter	Options (if any)	Meaning
funit		PRIMOS file unit number on which this file's segment directory is open. This specifies the MIDAS file that is being used.
buffer		Buffer into which retrieved data is stored.
key		Full primary or secondary key used to identify the data entry to be locked.
array		The 14-word block used for file handler/user communication. For direct access only, the parameter is used to input the user-supplied data entry number and entry size.
flags		Options that specify use of various parameters. The options for LOCK$ are described in the following table entries.
	FL$USE	For direct access, may be zero. Array is used anyway for direct access. For keyed index, 1 = use contents of array. Array must be set up from a previous successful call to one of the MIDAS file handler routines (have a condition code of 0 indicating successful retrieval). If the condition code is 1 indicating successful retrieval, there may be further synonyms. By supplying array, the same data that was retrieved from a previous call (FIND$, NEXT$, etc.) is obtained again; however, array must be correctly set up. If index indicates direct access, array must be set up with Entry size and Entry number.
	FL$SEC	Ignored

	FL$KEY	Include primary key in data returned to caller (unless specified, it is not returned). If not Ø, plenth must be specified with the value of the primary key in mind.
	FL$UKY	Ignored
	FL$RET	Return full array - For LOCK$ this bit must be SET for use by a following call to UPDAT$.
altrtn		An alternate return to be taken in the event an error occurs.
		Index to be used:
		Value — Meaning
		0 — Primary
		1 — Secondary
		2 — Tertiary
		...
		-1 — Direct Access
file no.		0 or -1
plenth		Length of Data to be input. A value of 0 means specify the full length of data entry file plus full Primary Key, if specified by FL$KEY.
keylnt		Ignored (see Note)

Using LOCK$

The following example retrieves and locks a record for updating on primary key.

```
        INTEGER *2 FLAGS, BUFFER (80), PKEY (6), FUNIT, ARRAY (14)
        EQUIVALENCE (BUFFER (1), PKEY (1))
        .
        .
        .

        FLAGS = FL$KEY + FL$RET
        CALL LOCK$ (FUNIT, BUFFER, PKEY, ARRAY, FLAGS, $9000,0,0,0)
        .
        .
        .
```

This example retrieves and locks a data record for updating using an array returned by NEXT$ on a secondary key:

```
        .
        .
        .
        INTEGER *2 FLAGS, BUFFER, PKEY (6), SKEY-2 (4), FUNIT, ARRAY (14)
        EQUIVALENCE (BUFFER (1), PKEY (1))

        .
        .
        .
        FLAGS = FL$RET + FL$USE + FL$UKY + FL$PLW + FL$KEY
  100   CALL NEXT$ (FUNIT, BUFFER, SKEY-2, ARRAY, FLAGS, $9000,2,0,0,2)

        .
        .
        .
        CALL LOCK$ (FUNIT, BUFFER, SKEY-2, ARRAY, FLAGS, $9100,2,0,0)
```

NEXT$

NEXT$ locates the next sequential data entry in the file. NEXT$ can be used to obtain the first entry in a sequence as well as subsequent entries and reads into buffer the specified amount of data. This routine is useful for ordered scans of MIDAS files according to ascending key order in one of the indexes. This subroutine ignores any data entry lockouts that may have been applied. Where synonyms are present on the file (secondary indexes only), NEXT$ is the only method of returning further synonyms. Repeated calls to NEXT$ retrieve synonyms in the time-order in which they were added to the file. NEXT$ cannot use the direct access method (i.e., index set to -1).

Calling Sequence

NEXT$ is called using the standard call. The parameter values involved in the call to NEXT$ are used to specify the various options available:

1. Access Methods
2. Information retrieved
3. Use of user-supplied array

The parameter settings for each option are discussed in the following paragraphs.

Access Methods

NEXT$ traverses the MIDAS file using the keyed index method only. Specification of the direct access method is meaningless and, therefore, illegal (i.e., an index value of -1 is illegal.)

Principles of NEXT$

1. FL$USE RESET indicates "get first entry as specified".
2. FL$USE SET indicates "get next entry according to array supplied".
3. FL$RET RESET is invalid.

Thus, NEXT$ may be used to position into the file. FIND$ may also be used if FL$RET is set for the FIND$ call. For NEXT$ to function after the first call, both FL$USE and FL$RET must be set in flags; the array must be used because it contains details of the current position in the index, and it must be returned for use in the next call to NEXT$. The first positioning call to FIND$ or first call to NEXT$ must return the array for use by the subsequent call to NEXT$.

If FL$USE in flags is not set, then NEXT$ behaves exactly like FIND$, i.e., it initiates an index search to locate the first data entry with a key that matches the full or partial key supplied.

Choice of Index

Either the primary or any of the 19 secondary indexes of a MIDAS file can be traversed sequentially by repeated calls to NEXT$. The index structure to be used is specified in parameter index. Values may be as follows:

Value	Meaning
-1	Illegal for call to NEXT$
0	Primary Index
1 to 19	Specified secondary index

When reading data entries in ascending key order of the chosen index, the user must supply the same index value in repeated calls to NEXT$.

The concept of next data entry means: the next data entry in a file with the next highest key value in that particular index. To alter the index parameter between repeated calls to NEXT$ is, therefore, meaningless.

Partial Key Values

Key values for NEXT$ are used in the following ways:

1. To initiate a sequential traverse of the index if FL$USE is reset.

2. To stop a sequential traverse of the index, if the key value is no longer satisfied when FL$PLW is reset.

The key value is used exactly as for FIND$; if a full key search is specified (say for duplicates) and FL$PLW is not set, an error code of 7 is returned if there are no entries with the specified key.

The key value and other parameters described below are used as selection criteria, on the next data entry. If the selection criteria are not satisfied, the next data entry is rejected and NEXT$ indicates to the user that the data entry is NOT FOUND.

FL$PLW in flags indicates the selection criteria. Its values are:

Value	Meaning
SET	Read the next data entry regardless
RESET	Read the next data entry only if it matches the user supplied full or partial key in parameter. Key for the length specified in KeyInt (in units specified by FL$BIT in flags). If the partial key does not match, NEXT$ indicates that the data entry is NOT FOUND.

As an example of the use of this selection criterion, consider a third secondary key with two alphabetic and three numeric characters. By using FIND$ and specifying the two character partial key AA, the user can position to the first key beginning with AA and retrieve the data entry. Thereafter, the user can retrieve in sequence each data entry that begins with AA. At the end of the entries beginning with AA, for example, the partial key changes to AB. NEXT$ takes the NOT FOUND exit if FL$PLW is reset. This indicates to the caller that the end of the AA's has been reached.

To position to the first data entry on the selected index, FL$FST in flags must be set. In this case the contents of key is immaterial. The entire file can be scanned, with data entries being returned according to ascending order of the chosen index. Since the indices supported are primary and 19 secondaries, a sort mechanism can be provided allowing 20 different sort keys.

Information Retrieved

The user options available in NEXT$ enable information to be retrieved from:

1. The Data Entry
2. The Primary Key value.
3. The Secondary Index Entry where more additional information can be stored to supplement the main data entry

In addition, user options enable this information to be returned to either the user-supplied buffer or the user-supplied key. The following paragraphs describe how the options are used to return information and where it is returned.

Use of Buffer

After a successful call to NEXT$, MIDAS returns the information from the next data entry corresponding to the index selected or, alternatively, information from the last level secondary index entry. This latter option is available only if a secondary index search is made. FL$SEC of flags must be SET to return information from the index entry rather than the data entry. Whichever option is chosen, the conditions described in the following paragraphs still apply.

The information retrieved can optionally be preceded by the corresponding primary key value returned from the data entry, regardless of whether the information was retrieved with a primary or secondary index. FL$KEY of flags must be set if this option is required. Buffer must be sufficiently long to contain all the specified return information, the data entry, and the full primary key, if specified.

If the user requires the return of all information, then plenth may be set to zero. Otherwise, plenth specifies the total number of words to be returned including the full primary key, if specified.

The facility to return the full primary key can be most useful to the user under the following example circumstances:

1. The next data entry has been found using a secondary index and it is required to know the value of the primary key value corresponding to that data entry.

2. The data entry has been found by supplying a partial primary key and it is necessary to identify the exact data entry accessed by the corresponding full primary key value.

3. The data entry has been found by supplying a secondary key, but the array indicates that further synonyms exist in the file. The identity of this first synonym can be established by reference to the full primary key returned, which is always unique.

Use of Key

The keyfield key usually provides to NEXT$ the key by which it optionally retrieves or rejects the next data entry from the file.

The full key corresponding to the next sequential data entry on the chosen index is returned to key if FL$UKY is set. Setting FL$UKY does not interfere with partial key values.

The user must ensure that the key parameter is large enough to contain the full key if FL$UKY is set.

Use of Array

The first word of array is always used to return completion codes or error codes to the caller, as follows:

Code	Meaning
0	Successful retrieval
1	Successful retrieval, but there may be further synonyms of the file.
	Error code.
Other	See Appendix A

When using NEXT$, FL$RET must be set in flags. If a call to NEXT$ is made without specifying FL$USE, NEXT$ is identical to FIND$, i.e., it searches the indexes using the keyed index access method and locates the first data entry that matches the full or partial key supplied (unless FL$PLW is set). FL$USE must be set to retrieve subsequent entries.

Summary of Parameters for NEXT$

Table 9-7 summarizes parameters and options for NEXT$.

Table 9-7. Summary of Parameter Values for NEXT$

Parameter	Options (if any)	Meaning
funit		PRIMOS file unit on which the file's segment directory is open. This specifies the MIDAS file being used.
buffer		Buffer into which retrieved data is stored.
key		Full or partial primary or secondary key to be used to identify the data.
array		14-word block used for MIDAS file handler and user communication.
flags		Options that specify use of various parameters. The options for NEXT$ are described in the following table entries.
	FL$USE	Use contents of array. This bit generally is set. Array must be set up by a previous call to FIND$ or NEXT$ for the same index. Direct access is not supported by NEXT$. If FL$USE is not set, then a first time search for the key supplied is made, similar to keyed index FIND$ operation.
	FL$RET	Must be set. Returns contents of Array.
	FL$KEY	Include primary key in buffer.
	FL$BIT	Keysize is specified in units or Bytes.
	FL$PLW	Get next entry regardless of key match.
	FL$UKY	Update keyfield with full key from data base.
	FL$SEC	Return secondary data from secondary index.

	FL$FST	Position to first index entry.
altrtn		An alternate return to be taken if an error occurs.
index		Index to be used. Its values are: Value — Meaning 0 — Primary 1 — Secondary n — n th index Note: Direct access is illegal.
file no.		0 or -1
plenth		Length of data to be input. Plenth = 0 means full amount in data entry plus full primary key, if specified.
keylnt		Length of key to be used and whether the user wants a full or partial key. Possible values are: Value — Meaning 0 — Full size key in data base nonzero — User length in words or bytes/bits as specified by FL$BIT

Using NEXT$

The following example traverses the whole file on Index 2:

```
        INTEGER *2 FLAGS, BUFFER (80), SKEY-2 (4), PKEY (6),
        ARRAY (14), FUNIT
        EQUIVALENCE (BUFFER (1), PKEY (1))
        .
        .
        .

        FLAGS = FL$RET + FL$FST + FL$KEY + FL$UKY
        CALL NEXT$ (FUNIT, BUFFER, SKEY-2, ARRAY, FLAGS, $9000,2,0,0,0)
        FLAGS = FL$USE + FL$RET + FL$KEY + FL$UKY + FL$PLW
        GO TO  1000

  100   CALL NEXT$ (FUNIT, BUFFER, SKEY-2, ARRAY, FLAGS, $9100,2,0,0,0)
 1000               (User)
        .
        .
        .

        GO TO 100
```

The use of NEXT$ does a partial key search on 3 characters of the Primary key:

```
        INTEGER *2 FLAGS, BUFFER (80), PKEY (6), ARRAY (14), FUNIT
        EQUIVALENCE (BUFFER (1), PKEY (1))
        .
        .
        .

        FLAGS = FL$USE + FL$RET + FL$BIT + FL$KEY
        ARRAY (1) = -1
  100   CALL NEXT$ (FUNIT, BUFFER, PKEY, ARRAY, FLAGS, $9000,0,0,0,3)
        .
        .
        .

        GO TO 100
```

This example gets all occurrences of synonym keys in secondary index 3:

```
      .
      .
      .
      INTEGER *2 FLAGS, BUFFER (80), PKEY (6), ARRAY (14), FUNIT
      REAL RKEY-3
      EQUIVALENCE (BUFFER (1), PKEY (1))

      .
      .
      .
      FLAGS = FL$RET + FL$USE + FL$KEY
      CALL FIND$ (FUNIT, BUFFER, RKEY-3, ARRAY, FLAGS $9000,3,0,0,0)

1000  .
      .
      .
      IF (ARRAY (1) NE. 1) GO TO 3000
      CALL NEXT$ (FUNIT, BUFFER, RKEY-3, ARRAY, FLAGS, $9100,3,0,0,0)
      GO TO 1000

3000  .   (USER PROCESS)
      .
      .
```

UPDAT$

UPDAT$ writes a replacement data entry from the caller's buffer.

The routine LOCK$ must have been called immediately before calling UPDAT$ and LOCK$ must have returned valid array contents for use by UPDAT$. (Condition code set to 0 or 1.)

Updates may be performed through any index, whether the file is keyed index or the direct access.

Calling Sequence

UPDAT$ is called using the standard call described previously in this section. The parameter values involved in the call to UPDAT$ specify the options available in the following three areas:

1. Access Methods
2. Information Retrieval
3. Use of User supplied Array

The parameter settings for each option are discussed in the following paragraphs.

Access Methods

There are two access methods supported by UPDAT$. These are:

1. Keyed Index
2. Direct Access

The basic access method is keyed index. The direct access method is potentially faster than the keyed index method for retrievals.

The same index value used to LOCK the record must be used to update it.

Value	Meaning
-1	When direct access is a specified option for this file.
0	When direct access is not specified for this file.
n	Any secondary index.

Partial keys are not relevant to UPDAT$, and the parameter kevlnt is ignored. Since array must be supplied, FL$USE in flags must be set.

Information Returned

If FL$ULK of flags is set, UPDAT$ only unlocks the data entry and does not update it. Otherwise, the complete updated data entry must be present in buffer.

The data entry may optionally be preceded by the full primary key in buffer, in which case FL$KEY in flags must be SET.

Use of Array

Array must be supplied on input (FL$USE set in flags) and must contain information returned from the previous successful LOCK$ call.

If FL$RET of flags = 1 the return of an array takes place, but it is meaningless since, after a successful call, the contents are not changed.

The first word of the retained array is always used to return completion codes or error codes to the caller, regardless whether FL$RET of flags is SET. These codes are:

Code	Meaning
0	Successful update
11	Entry not locked by user; unsuccessful update
other	Error code; see Appendix A

Summary of Parameters for UPDAT$

Table 9-8 summarizes parameters and options for UPDAT$.

Table 9-8. Summary of Parameter Values for UPDAT$

Parameter	Option (if any)	Meaning
funit		PRIMOS file unit number on which this file's segment directory is open. This specifies the MIDAS file being used.
buffer		Buffer in which the updated data entry must be presented. It may include Primary Key.
key		Full or partial primary or secondary key.
array		Must be supplied
flags		Options that specify use of various parameters. The options for UPDAT$ are described in the following table entries.
	FL$USE	User array must be set
	FL$RET	Return array, may be set. However, this call is meaningless, and array is not set after a call to UPDAT$.
	FL$KEY	If set, there is a full primary key in buffer.
	FL$ULK	If set, unlock only, do not update data. All other bit settings for flag are ignored.

Using UPDAT$

The following example updates a record previously locked through index 2:

```
.
.
.
INTEGER *2 FLAGS, BUFFER (80), PKEY (6), ARRAY (14), SKEY-2 (4)
FUNIT
EQUIVALENCE (BUFFER (1), PKEY (6))

.
.
.
FLAGS = FL$USE + FL$KEY
CALL UPDAT$ (FUNIT, BUFFER, SKEY-2, ARRAY, FLAGS, $9100,2,0,0)
.
.
.
```

SECTION 10

MODIFYING MIDAS TO MEET USER NEEDS

MIDAS parameter values that users may have a legitimate need to change have been pulled out of the collection of MIDAS source modules and isolated for the most part in two modules. The first of these is a parameter file called in the UFD named SYSCOM called KPARAM. The second is a dummy routine for defining COMMON called LDPOOL. This section describes the various parameters in SYSCOM>KPARAM and LDPOOL and under what circumstances their values should be changed.

LDPOOL, COMMON CONTROL MODULE

LDPOOL contains the basic definition of the common areas for the MIDAS on-line file handler. Part of the on-line file handler is loaded with almost every MIDAS program regardless of whether it is a program for use in a multi-user environment or is intended to be run by a single user in off-line environment. Accordingly, a version of LDPOOL is loaded with every MIDAS program. There is a version built into the libraries named KIDALB, VKDALB, and KIDAFM by the command files used to generate these libraries. This version may be replaced by a user-created version by loading it before KIDALB. The definition of the common areas has been handled in this way so that the common areas may easily be tailored for each application without requiring that any of the MIDAS libraries be rebuilt.

LDPOOL defines the on-line buffer pool POOLBG and the control array for POOLBG, CTLA. POOLBG is the destination array for all index blocks that MIDAS needs to examine. The standard rule of thumb for the length of POOLBG is that it must be long enough to hold two of the longest last-level index blocks likely to be encountered, plus a little extra. POOLBG must be long enough to hold at least one of any size index block. The provision for a little extra is so that most first level index blocks can be held in memory through a MIDAS call. A buffer pool that is twice as long as the longest last-level index block can hold both a main index last-level block and an overflow last-level block during the course of a single call to MIDAS, which improves the efficiency of NEXT$.

As delivered, POOLBG in KIDALB is 5000 octal locations long, based on the assumption that the longest index block will be 1024 (2000 octal) words long from a storage module. This is the default index block size for Rev 14.0. Previous versions of KIDALB defined POOLBG to be 2000 octal (1024) words long assuming a maximum index block size of 440 words. Users modifying the default index block size or using the full options stream of CREATK to define all block sizes as less than 1024 words long may wish to shorten the length of POOLBG accordingly. It is suggested that 2-1/2 times the longest last level index block is an appropriate size (but POOLBG must be at least as long as the longest index block of any size).

To control the usage of POOLBG, there is an array CTLA also defined in LDPOOL. As delivered, LDPOOL in KIDALB defines CTLA to hold information for 19 index blocks. When CTLA is filled (all 19 positions used), old positions in POOLBG are used, destroying index blocks previously in memory. Hence, CTLA should be large enough to make full use of POOLBG in any conceivable situation. On the other hand, five words are required to retain information for each index block in memory. A user with space problems in memory may wish to cut down the number of positions in CTLA based on knowledge of the actual sizes of index blocks in his files.

The final array with a user-variable size in LDPOOL is FILES. FILES is designed to contain information on MIDAS files when it is desired to retain an index in memory from one MIDAS call to another. Previous releases of MIDAS documented calls to OPEN$ and CLOSE$ which activated this feature. These calls are still available. However, their use is not recommended because retaining index blocks in memory, as presently implemented, will substitute paging activity for regular disk activity. Users not planning to retain index blocks in memory from one MIDAS call to the next may set the file size parameter in LDPOOL to 0. As delivered, KIDALB supports retention of information for 10 MIDAS files.

VKDALB is built with a longer version of POOLBG on the assumption that programs running in V-mode on the Prime 400 have more memory available and can benefit from the increased efficiency of a larger buffer pool, without significantly increasing paging. At Rev 14, the default size of POOLBG built into VKDALB is 10240 (24000 octal) locations. This version of POOLBG is defined in the modified copy of LDPOOL called LONGPL in the UFD named KI/DA on the master disk. CTLA is set up to hold information about 99 index blocks in LONGPL and FILES will hold information for about 20 MIDAS files.

There is a second special purpose version of LDPOOL called MIDPOL that is designed for use with off-line programs only. POOLBG contains 28674 words; CTLA supports 39 entries and FILES supports 0 files.

Unless either LONGPL or MIDPOL is exactly what is desired, the user should start with a copy of LDPOOL and create his own version rather than modifying the copies in the UFD named KI/DA. If the changes are to be made within the libraries themselves, then the copies in the UFD named KI/DA must be changed.

Each of the arrays POOLBG, CTLA and FILES is determined by a parameter in LDPOOL. These are:

POOLSZ - Size of POOLBG

CTLASZ - Number of entries in CTLA

FILESZ - Number of entries in FILES

To change the size of any or all of the arrays, the user need only change the values of these parameters and recompile a new version of LDPOOL. This new version can then be loaded before KIDALB, VKDALB or KIDAFM to implement the new sizes for a particular application. If LDPOOL or LONGPL are changed and KIDALB, etcetra is rebuilt with the new values, then the changes are global across all applications loaded with MIDAS.

KPARAM, MIDAS PARAMETER FILE

The parameters in KPARAM are used by almost all MIDAS routines. If values in KPARAM are changed, the user's MIDAS libraries must be completely rebuilt. Any applications that require the change must be loaded with the new MIDAS library. All the command files to build the MIDAS utility programs must also be run.

KPARAM is divided into two sections separated by a line of asterisks. Parameters below the line must not be changed by users. Parameters above the line may be changed according to the suggestions outlined in this section. Any changes made to parameters above the line of asterisks will not affect the ability of the rebuilt MIDAS libraries to access existing files. Changes made to KPARAM do not change the size or shape of existing MIDAS files, unless a new version of CREATK is used to introduce the changes.

If, for example, a user is planning to move a MIDAS file from a 60 megabyte disk to a 300 megabyte storage module and wishes to optimize the file for access on the storage module, the Rev. 14 version of CREATK should be run on the file. The Extend function must be run with a CR (return) entered in response to the SEGMENT DIRECTORY LENGTH and SEGMENT LENGTH queries. This tells CREATK to use its default values. Then, each index (including the primary index) must be accessed with CREATK's Modify command and a CR entered in response to the BLOCK SIZE query. (Or, minimum options can be selected.) In this way, when the file is next restructured by REMAKE it is configured for best access on the storage module.

RECLNT Default Index Block Length

The optimum size for an index block is an integral multiple or fraction of one physical disk record. For example, if the physical disk record is 440 words, suitable block sizes are 880 words, 440 words, 220 words, etc. If the physical block size is 1024 words, suitable blocks sizes are 1024 words, 512 words or 256 words.

The user may wish to increase the default block size if all his indexes are very long. A larger block size will make index access the indices more efficient. On the other hand, the user may wish to decrease the default block size if all his indexes are small and program size is a problem. If the index block size is smaller, then POOLBG can be smaller.

The maximum allowable block size is 1024 words. It is not recommended that block sizes smaller than 220 words be used.

As delivered, RECLNT is set to 1024 words, which assumes a storage module disk. If the user has no storage modules, RECLNT may be changed to 440 words (or 880 or 220 as indicated above).

Index block sizes may be changed on a per-file basis by using the full options stream of CREATK.

SEGLNT - Length of the Segment Directory

The length of the MIDAS segment directory is arbitrarily set to one physical disk record on a storage module. If the length is shorter, the data subfile will not be able to hold as many records, but the PRIMOS commands FUTIL and MAGSAV/MAGRST will be able to deal with the file more efficiently. If the length of the segment directory is longer, the data subfile will hold more records, but FUTIL and MAGSAV/MAGRST will be slower.

The user may want to reduce SEGLNT to 440 words if there are no storage modules on the system. If the data subfile size is only a problem for some files on a system, the segment directory size may be changed on a per-file basis by CREATK.

IWRAP - Number of Words per Segment

The length of a segment is expressed in 16-bit words rather than in terms of records and words. The length of a segment is arbitrarily set to enable one level of indexing to manage up to 524288 words in a DAM file on a storage module. On any other disk, this length requires two levels of indexing in the DAM file directory. Users that do not have storage modules may prefer to change this value to 193600 words, which on all other disks is the length of file which can be accessed as a DAM file with only one level of directory indexing.

Regardless of the disk type on the system, users may wish to change IWRAP to a larger number to increase the number of entries that files can hold. For example, doubling the length of a segment doubles the number of entries that each index can hold and also doubles the number of data records which the data subfile can hold. Of course, disk space must be available too, for the file is also capable of using twice as many disk records.

If file size is a problem only with some of the files on the system, the length of a segment can be changed on a per-file basis with CREATK.

BREAKE - Program Interrupt Control

During calls to ADD1$ and between calls to LOCK$ and UPDAT$, MIDAS disables the user program interrupt capability. That is to say the user may not interrupt the running of the program with control-P or Break. This is done to guarantee file integrity. At the end of the call to ADD1$, and after the call to UPDAT$, MIDAS re-enables the program interrupt capability. Some users may wish to prevent MIDAS from re-enabling Break. This assumes that the user is, instead, going to call BREAK$, at a time of the user's choice.

The parameter BREAKE controls MIDAS calls to re-enable Break. As delivered BREAKE is set to 1. The value has the effect of causing MIDAS calls to BREAK$ to disable and enable the program interrupt facility as indicated above.

To prevent MIDAS from re-enabling the program interrupt facility, set BREAKE to 0. This causes MIDAS to disable breaks as usual but prevents it from re-enabling them. The user may then re-enable them or not as desired.

RECYLA-Recycle Control Parameter

When attempting to open a segment that some other user may have open, MIDAS calls the PRIMOS file system subroutine RECYCL and tries again to open the segment. This is done the number of times specified in parameter RECYLA. If RECYCLA is not large enough, MIDAS may, from time to time, return to the user with an error code 22 or 24. This problem is most likely to occur on a Prime 400 with only two or three users. The problem may go away if RECYLA is increased. As delivered, RECYLA is set to 100.

KFILE, Start of MIDAS'S File Units

MIDAS uses three of the 16 available PRIMOS file units for accessing segments under the segment directory. As delivered these are file units 14, 15, and 16. Some users may wish to have MIDAS use other file units. The three file units must be contiguous. Within this limitation, the user may choose another set of file units by changing the value of KFILE. For example, if KFILE is set to 7, then MIDAS will use file units 7,8 and 9.

BIGSIZ, Default Number of Entries for PRIBLD and SECBLD

To build a MIDAS file using PRIBLD and SECBLD without specifying the number of entries in the file, or to REPAIR a file, a default number of entries for the file is assumed. This value is defined by BIGSIZ. As delivered BIGSIZ is set to 200000. If this is not large enough, BIGSIZ may be set to a larger value.

OVFCON - BILD$R Control

When building an index with BILD$R, the new records are added to overflow. At intervals the overflow is merged into the main body of the index. BILD$R computes an optimum value for the maximum number of blocks in an overflow chain before the merge is done. However, there is a minimum number below which it is not worth doing the merge. The minimum number is defined by OVFCON. As delivered, OVFCON is set to 7. It is possible that this value may not be optimum for all index block sizes. A user running programs calling BILD$R frequently may wish to set OVFCON higher or lower. The optimum value must be determined by experiment.

IBULEN - Buffer Pool Size

The MIDAS utilities REMAKE and REPAIR use a buffer pool to build index blocks in and to hold control information about the size and number of index blocks. The size of this buffer pool is determined by the value of IBULEN. As delivered, IBULEN is set to 15000. Users making several concurrent calls to PRIBLD, SECBLD and BILD$R may find that this value is not large enough. Other users concerned about program size may find that a smaller value will do. The value of IBULEN may be changed by the user according to the following guidelines:

1. For building an index with PRIBLD or SECBLD, allow 86 words plus the length of an index block for each level of indexing expected in each index to be built.

2. For one or more indices to be built with BILD$R, allow 6006 words.

3. For a fixed length data record, allow the length of the data plus the length of the primary key plus 2.

4. For a variable length data record, allow 4096 words plus the length of the primary key plus 2.

5. Add at least 100 words for other small buffers.

Suppose a MIDAS file has the following parameters:

- Primary key - 10 words
- Data length - Fixed 120 words
- Secondary index subfiles 1,2,3, and 4
- Data stream sorted on the primary key
- Allow 5 levels of indexing for all indices
- Assume all index blocks 1024 words long

The primary index and data will be built with PRIBLD, all the secondaries with BILD$R. Then PRIBLD will require 5120+86 words. BILD$R will require 6006 words and the data record will require 132 words. Allowing 100 extra words, IBULEN must be at least 12244 words.

If the incoming data stream were sorted on two secondary index keys as well as the primary index and all other factors remained the same, 5120+86 words (6006) would have to be allowed for each of these secondary indices as well. IBULEN would then have to be at least 12244+12012 or 24256.

OFFSIZ-Size of the Off-Line-On-Line Buffer Pool

Off-line programs, in particular REMAKE, KBUILD, and REPAIR, are loaded with MIDPOL to give a larger POOLBG for increased speed. User off-line programs may be loaded with MIDPOL also for the same reason. The size of POOLBG in MIDPOL is determined by the parameter OFFSIZ. As delivered OFFSIZ is set to octal 70002. Users may find that OFFSIZ could be even larger. OFFSIZ only affects the utility programs mentioned above plus MIDPOL. If the value of OFFSIZ is changed, only these utilities must be rebuilt. The MIDAS libraries do not need to be rebuilt.

MKEYSZ-Maximum Key Size for ASCII Keys

As delivered, the maximum key size for ASCII keys is set to 64 characters (32 words). The size of various buffers is computed based on this length. If a user wishes to use longer ASCII keys, the length of MKEYSZ may be increased appropriately and the MIDAS libraries rebuilt.

SECTION 11

EXAMPLES

This section presents examples of how MIDAS capabilities can be used to build and access a MIDAS file. The example below is written in FORTRAN since COBOL and RPG use the protocols for those languages.

HYPOTHETICAL USER EXAMPLE

First, assume a hypothetical user application that will use a MIDAS file with two secondary index subfiles. For demonstration purposes, one of these secondary index subfiles will contain secondary data along with the secondary index entries. The other secondary index will not contain secondary data.

The purpose of the application is to maintain an up-to-date customer file for a national manufacturing concern. The file contains:

- at least one data record for each customer.
- information about the status of each account.
- backorder information.

Each data record will be 120 words long initially. The customer wishes to be able to increase the size of the data record to 150 words at some future date without impacting existing applications.

A backorder occurs when a customer places an order with the manufacturer that cannot be completely filled. If acceptable to the customer, the unfilled portion of the order is placed on backorder, and when the items become available, the backorder is filled on a first-come-first-served basis.

The manufacturer has a central computer in its home office. All sales orders are entered into this central computer by an interactive program that can be accessed by phone lines from all the remote sites. Hence, one file is maintained for all the company's customers.

The manufacturer has a different division for manufacturing each different product. There is a national sales staff that sells the products of all divisions, divided into regions of the country. Customer orders are always directed to a particular division of the company. That is to say, a customer purchasing from more than one division of the company purchases from each as if they were separate companies. Most customers can be handled by one regional office. Several customers are national concerns and thus are serviced by more than one regional office. In such cases, the manufacturer considers each region as having a separate account with the customer.

Primary Index

Each customer is identified by a unique alphanumeric 12-character customer ID. broken up into three fields. The first eight characters are numeric and identify the customer. The next two characters identify the division with which the customer does business. The last two characters identify the regional office handling the order. These last two fields are alphabetic. This 12-character ID is the primary key for the file.

First Secondary Index

In some instances the manufacturer found that it was easier to have access to the customer file through customer name and address alone. The name and address field in each customer record is 80 characters long. However, a hashing algorithm was applied to compress the name and address fields into a single seven-word field. The hashing algorithm does not guarantee a unique identifier for each customer but does locate the general area in the customer file from which the particular customer record can be located. Additional accesses through a sequential traverse of the file can locate the particular record. This is the key for the first secondary index. This index has no secondary data stored with the index entry.

Second Secondary Index

The backorder information is stored in the second secondary index using a two-word key. Each item that the manufacturer sells has a four-digit part number. This becomes the first word and is stored as an integer value. The second word contains the date the customer's order was placed (a five digit number referenced from a base date) and also stored as a integer. The secondary data that is stored with each index entry comprises the quantity of the item on backorder, the ID for the regional sales office handling the account, and salesman's ID. This requires three additional words for each secondary index entry.

USING CREATK TO BUILD A TEMPLATE

The minimum options stream of CREATK was used. If, in the future, the optimizing features of CREATK are required, each index can be re-shaped using the full options stream. Refer to Section 2 for details of the CREATK dialog. Figure 11-1 shows the steps used in creating the template for the example file and shows how CREATK was reinvoked to specify new characteristics for the secondary index subfile 2.

Primary Index and Data

The customer ID in the primary key is in characters; therefore, the designer chose ASCII for the key type. It is easier to consider the key length in characters also; therefore, the key length was given as B 12.

```
OK, CREATK
GO
MINIMUM OPTIONS? YES

FILE NAME? CUSFIL
NEW FILE? YES
DIRECT ACCESS? NO

DATA SUBFILE QUESTIONS

KEY TYPE: A
KEY SIZE = : B 12
DATA SIZE = : 12Ø

SECONDARY INDEX

INDEX NO.? 1

DUPLICATE KEYS PERMITTED? YES
KEY TYPE: A
KEY SIZE = : W 7
USER DATA SIZE = :

INDEX NO.? 2

DUPLICATE KEYS PERMITTED? NO
KEY TYPE: B
KEY SIZE = : W 2
USER DATA SIZE = : 3

INDEX NO.?

OK, CREATK
GO
MINIMUM OPTIONS? YES

FILE NAME? CUSFIL
NEW FILE? NO
FUNCTION? MODIFY

INDEX NO.? 2
DUPLICATE KEYS PERMITTED? YES
USER DATA SIZE = : 3


INDEX NO.?

FUNCTION? QUIT

OK,
```

Figure 11-1. CREATK Dialog to Define and Modify Customer File

Although expansion of the data record to 150 words is anticipated, this expansion can be accomplished at a later date by increasing the data size using CREATK and then invoking the REMAKE program to restructure the file. With this in mind the applications designer chose to set the original data size to 120 words.

Secondary Index 1

Because the value of this key is not expected to be unique, duplicate keys are supported for this index.

The key value resulting from the hashing algorithm is in characters so the key type is ASCII. As indicated, the designer thinks of this key as being in words; therefore, the key length was given as W 7.

There is no secondary data for secondary index-1 so the response to USER DATA SIZE was a CR to indicate Ø.

Secondary Index 2

At first, the designer thought that this index would have no need to support duplicate entries, so the original MIDAS template was not built to permit duplicate entries in this field. Almost immediately, this was discovered to be the wrong choice. The existing file was made usable by rerunning CREATK and changing the option to support duplicate entries. REMAKE was invoked to restructure index 2 (See Figure 11-2), and the applications program was immediately able to create duplicate entries as necessary.

Consideration was given to whether this secondary index key should be declared as a long integer or as a bit string. When a part on backorder became available it would be necessary to locate an entry in the index based on the part number alone. This requires that a partial key search be done on the first word of the key, whoch would not be possible if the key were considered a long integer. Consequently, the key was declared a Bit String key type. The length, two words, was given as W 2.

In this case, there are three words of user data. The answer to SECONDARY DATA SIZE, therefore, was 3.

```
OK, LISTF

UFD=MIDEXP 5 ,

CUSFIL OLDFIL LLXXØ1

OK, REMAKE
GO

FILE NAME? CUSFIL
INDICES? 2

INDEX SUBFILE     2

ENTRIES INDEXED:     Ø
ENTRIES IN OVERFLOW:     Ø
ENTRIES DELETED:     Ø
TOTAL ENTRIES IN FILE:     Ø

ENTRIES INDEXED:     Ø
INDEX LEVELS:     1

OK,
```

Figure 11-2. Using REMAKE to Restructure Modified File

BUILDING THE INITIAL FILE WITH KBUILD

As the MIDAS application replaced existing applications using data from tape and disk files, the applications designer proceeded as follows:

1. Wrote a conversion program to rearrange the data in the sequential source files into the proper MIDAS data record image.

2. Used KBUILD to process the converted files into MIDAS files.

The dialog in Figure 11-3 shows the use of KBUILD to build the example file. A labelled FUTIL listing of the file shows the file structure resulting from the invokation of CREATK and KBUILD in this example.

The decision to perform the conversion in two passes was made because the programming staff was already familiar with the data sources and could go to work immediately. If they had stopped to learn the MIDAS file handler subroutines in section 10, they would have been able to handle the conversion in one pass. However, the penalty would have been a greatly increased number of man hours required to complete the project.

KBUILD was used to build the primary index and secondary index 1. A one-to-one correspondence does not exist between the entries in secondary index 2 and the data file so this index could not be built with KBUILD. Being a small index it was easily built using an applications program designed to manipulate the file in general.

The input data record produced by the conversion and processed by KBUILD contained 133 words. The first 12Ø were the MIDAS data record image, the next six were the primary key, and the final seven were the hash of the name and address fields (the key for secondary index 1). The program used to build the sequential file called PRWFIL to write the data to disk, as some of the data records were not in ASCII characters. None of the keys are part of the MIDAS record although the applications program always restores the MIDAS copy of the primary key with the data record.

```
LISTF

UFD= MIDEXP 5 0

CUSFIL OLDFIL

OK, KBUILD
GO
SECONDARIES ONLY? NO
ENTER INPUT FILE NAME: OLDFIL
ENTER INPUT RECORD LENGTH (WORDS) : 133
INPUT FILE TYPE: TEXT
ENTER NUMBER OF INPUT FILES: 1
ENTER OUTPUT FILE NAME: CUSFIL
ENTER STARTING CHARACTER POSITION, PRIMARY KEY: 121
SECONDARY KEY NUMBER: 1
ENTER STARTING CHARACTER POSITION: 127
SECONDARY KEY NUMBER:
IS FILE SORTED? NO
ENTER LOG/ERROR FILE NAME: LLXXØ1
ENTER MILESTONE COUNT:
OLDFIL
COUNT     DATE      TIME      CPU MIN    DISK MIN    TOTAL TM    DIFF
     Ø  Ø9-Ø1-77  14:38:Ø5    Ø.ØØØ       Ø.ØØØ       Ø.ØØØ     Ø.ØØØ

INDEX SUBFILE   Ø
ENTRIES INDEXED:          2
INDEX LEVELS:   1

INDEX SUBFILE   1
ENTRIES INDEXED:          2
INDEX LEVELS:   1

INDEX SUBFILE   2
ENTRIES INDEXED:          2
INDEX LEVELS:   1
END OF RUN
     5  Ø9-Ø1-77  14:38:21    Ø.Ø25       Ø.Ø28       Ø.Ø53     Ø.Ø13

OK,
```

Figure 11-3. Sample KBUILD Dialog

```
OK, FUTIL
GO
  LISTF 3 TYPE SIZE
FROM-DIR = *
TO-DIR = *

BEGIN *                    1 UFD

OLDFIL        1 SAM

BEGIN CUSFIL               2 SEGSAM        MIDAS file segment directory

(     0)      1 SAM        (   1)  3 SAM   File Descriptor Subfile
(    11)      3 SAM        (  21)  3 SAM
(   185)      1 DAM                        Primary Index Subfile

END CUSFIL                 13 SEGSAM       Secondary Index 2 Subfile
                                           Secondary Index 1 Subfile
LLXX01        1 SAM                        Data Subfile

END   *                    16 UFD
  QUIT

OK,
```

Figure 11-4. FUTIL Listing of Sample MIDAS File

USING THE ON-LINE MIDAS FILE HANDLER

Setting Up

When the data records had been converted and the initial applications designed, the designer create an $INSERT file called KIDINS to describe the data record and facilitate communication between the main program and various subroutines. Note that the designer allowed 150 words for the MIDAS data record in anticipation of the increased data record length. For example:

```
C     KIDINS
        INTEGER*2 FL$USE,FL$NXT,
      +  FL$RET,FL$KEY,FL$BIT FL$PLW,FL$UKY,FL$SEC,FL$ULK,FL$FST
C
C       MIDAS FLAGS VALUES
C
        PARAMETER FL$USE=:100000        /*USE ARRAY
        PARAMETER FL$RET=:40000 /*RETURN ARRAY
        PARAMETER FL$KEY=:20000 /*INCLUDE PRIMARY KEY IN BUFFER
        PARAMETER FL$BIT=:10000 /*KEY IN BITS OR CHARS
        PARAMETER FL$PLW=:4000  /*GET NEXT ENTRY, REGARDLESS
        PARAMETER FL$UKY=:2000  /*UPDATE KEY FIELD IN USER PROGRAM
        PARAMETER FL$SEC=:1000  /*RETRURN SECONDARY DATA
        PARAMETER FL$ULK=:400   /*UNLOCK, DON'T UPDATE
        PARAMETER FL$FST=:200   /*GET FIRST RECORD IN INDEX
        PARAMETER FL$NXT=:100   /*GET RECORD 'NEXT GREATER THAN'
C
        INTEGER DBUFFR(156),  /*BUFFER FOR DATA AND PRIMARY KEY
      +  PKEY(6),             /*BUFFER FOR PRIMARY KEY
      +  SKEY1(7),            /*BUFFER FOR SECONDARY KEY 1
      +  SKEY2(2),            /*BUFFER FOR SECONDARY KEY 2
      +  SBUFFR(9),           /*BUFFER - PRIMARY KEY, 2NDARY DATA
      +  NAME(11),            /*CUSTOMER NAME FIELD
      +  ADDR(26),            /*CUSTOMER ADDRESS FIELD
      +  ZIP(3),              /*CUSTOMER ZIP CODE
      +  CODE,                /*CUSTOMER CREDIT CODE
      +  BPKEY(6),            /*PRIMARY KEY COMES WITH 2NDARY DATA
      +  PARTNO,              /*PART NUMBER FIELD OF 2NDARY KEY 2
      +  BDATE,               /*DATE FIELD OF 2NDARY KEY 2
      +  PARTS,               /*NUMBER ON BACK ORDER, 2NDARY DATA
      +  REGION,              /*REGION, 2NDARY DATA
      +  SALSID,              /*SALESMAN'S ID, 2NDARY DATA
      +  ARRAY(14),           /*MIDAS COMMUNICATIONS ARRAY
      +  BARRAY(14)           /*EXTRA MIDAS COMMUNICATIONS ARRAY
        DOUBLE PRECISION BALANC   /*OUTSTANDING BALANCE
            .
            .                     /*ETC.
            .
```

```
      COMMON /KIDCOM/DBUFFR,SKEY1,SKEY2,SBUFFR,ARRAY,BARRAY
      EQUIVALENCE (DBUFFR,PKEY),/*PRIMARY KEY ALWAYS WITH DATA
     +  (NAME,DBUFFR(7)),        /*NAME FIELD IN DBUFFR
     +  (ADDR,DBUFFR(18)),       /*ADDRESS FIELD IN DBUFFR
     +  (ZIP,DBUFFR(44)),        /*ZIP CODE IN DBUFFR
     +  (CODE,DBUFFR(47)),       /*CREDIT CODE IN DBUFFR
     +  (BALANC,DBUFFR(48)),     /*OUTSTANDING BALANCE IN DBUFFR
              .
              .                 /*ETC.
              .
      EQUIVALENCE (SKEY2,PARTNO), /*MANUFACTURERS PART #
     +  (SKEY2(2),BDATE),        /*DATE OF BACKORDER
     +  (SBUFFR(1),BPKEY),       /*PRIMARY KEY FOR BACKORDER
     +  (SBUFFR(7),PARTS),       /*NUMBER ON BACKORDER
     +  (SBUFFR(8),REGION),      /*SALES REGION
     +  (SBUFFR(9),SALESID)      /*SALESMAN'S ID
          .
          .
          .
```

File Handler Calls

The method for handling error conditions is demonstrated in the first two examples. Thereafter, it is assumed that the user is aware of the technique for error handling. Throughout the examples, it is assumed that the MIDAS file was opened on File Unit 1. This can be accomplished by a call to SEARCH, for example:

```
CALL SEARCH (1,'KIDAFL',1)
```

Locating Data Record By Primary Key Using FIND$

One of the purposes of the application was to enable members of the sales staff to interrogate customer data records. It is assumed that the staff member knows the customer ID. In this portion of the application, updates are not permitted; hence, it is not necessary to be able to re-access the record. The application asks the user for the appropriate primary key value which is read into PKEY. A routine called DISPLY is called to display the record if it is found. This call to FIND$ is located in a subroutine called FINDIT. The part of FINDIT of interest to MIDAS follows:

```
      SUBROUTINE FINDIT(.........
$INSERT KIDINS
      INTEGER FLAGS
      DATA FLAGS/FL$KEY/      /*RETURN PRIMARY KEY WITH DATA
              .
              .
              .
```

```
C
C   DATA GOES INTO DBUFFR, INCLUDING MIDAS'S COPY OF PRIMARY KEY
C   USER HAS PUT COPY OF PRIMARY KEY INTO PKEY FOR CALL
C   FLAGS TELLS MIDAS TO RETURN PRIMARY KEY
C   INDEX IS Ø - FIRST Ø IN ARG LIST
C   FILE# IS Ø - SECOND Ø IN ARG LIST
C   PLNGTH IS Ø - 3RD Ø IN ARG LIST, SAYS RETURN FULL RECORD
C   KEYLNT IS Ø - 4TH Ø IN ARG LIST, SAYS USE FULL KEY
C
1ØØØ   CALL FIND$(1,DBUFFR,PKEY,ARRAY,FLAGS,$9ØØØ,Ø,Ø,Ø,Ø)
       CALL DISPLY
       RETURN
C
C      THERE WAS AN ERROR
C
9ØØØ   IF(ARRAY(1).NE.7) GO TO 91ØØ
       WRITE 9ØØ1
9ØØ1   FORMAT('RECORD NOT IN FILE')
       RETURN
C
91ØØ   IF(ARRAY(1).EQ.22) GO TO 92ØØ
C
C      ALL OTHER ERRORS ARE FATAL!!!!
C
       CALL ERROUT              /*USER ROUTINE TO ABORT
92ØØ   DO 93ØØ I=1,1ØØ
       CALL RECYCL              /*A FILE WAS BUSY - WAIT A WHILE
93ØØ   CONTINUE
       GO TO 1ØØØ                       /*TRY AGAIN
       END
```

Using FIND$ to Start A NEXT$ Sequence

The backorder index is sometimes used by the staff to determine how many of a particular item, if any, are on backorder. If nothing is on backorder, the routine HOWMNY indicates this and returns immediately. If there are some parts on order, the number is summed and the value reported. It is assumed that the rest of HOWMNY has managed to set up the key value. Only the high order word (the part number) is supplied as the secondary key value. The low order word (date of order reference) is ignored, as the search is date independent.

```
      SUBROUTINE HOWMNY(.............
$INSERT KIDINS
      INTEGER FLAGS,RSUM
            .
            .
            .
```

```
C
C   FLAGS IS SET BELOW AS FOLLOWS:
C   PRIMARY KEY RETURNED IN BUFFER, ARRAY RETURNED IF SUCCESSFULL
C   SECONDARY DATA RETURNED - NOT BASIC DATA RECORD
C
       FLAGS=FL$KEY+FL$RET+FL$SEC
C
C   DATA WILL GO INTO INTO SBUFFR
C   USING SKEY2 (OR THE FIRST WORD THEREOF)
C   INDEX = 2, THE BACKORDER INDEX
C   FILE# = 0, AS USUAL - THE FIRST 0
C   PLNGTH - THE 2ND 0, SAYS RETURN ALL OF DATA
C   KEYLNT = 1, THIS MEANS 1 WORD OF THE KEY IS USED SINCE
C         FL$BIT IS NOT SET IN FLAGS.  THIS ACTIVATES THE
C         PARTIAL KEY SEARCH.
C
100    CALL FIND$(1,SBUFFR,SKEY2,ARRAY,FLAGS,$8000,2,0,0,1)
C
C   GOT ONE , SET UP COUNTER AND MOVE ON TO NEXT$
C
       RSUM=PARTS
       FLAGS=FLAGS+FL$USE              /*NOW USE ARRAY TOO
C
C   SEARCH WILL STOP WHEN PART NUMBER NO LONGER MATCHES THE
C   HIGH ORDER PART OF SKEY2 AS FL$PLW IS NOT SET IN FLAGS.
C   THE REST OF THE ARGS ARE THE SAME AS THE CALLS TO FIND$.
C
1000   CALL NEXT$(1,SBUFFR,SKEY2,ARRAY,FLAGS,$9000,2,0,0,1)
       RSUM=RSUM+PARTS                 /*ADD TO THE COLLECTION
       GO TO 1000
C
C   ERROR RETURN FROM FIND$
C
8000   IF(ARRAY(1).EQ.7) GO TO 8200
       IF(ARRAY(1).NE.22) GO TO 9500
       DO 8100 I=1,10
       CALL RECYCL                     /*A FILE WAS BUSY
8100   CONTINUE
       GO TO 100                       /*TRY AGAIN
C
8200   WRITE 8201
8201   FORMAT('NONE ON BACKORDER')
       RETURN
C
C   ERROR RETURN FROM NEXT$
C
9000   IF(ARRAY(1).EQ.7) GO TO 9300
       IF(ARRAY(1).NE.22) GO TO 9500
       DO 9100 I=1,10
       CALL RECYCL                     /*A FILE WAS BUSY
9100   CONTINUE
       GO TO 1000                      /*TRY AGAIN
```

```
C
C       THEY HAVE ALL BEEN COUNTED
C
9300    WRITE 9301, RSUM
9301    FORMAT(I5,' ON BACKORDER')
        RETURN
C
9500    CALL ERRCUT                          /*ABORT - ERROR WAS FATAL
C
        END
```

Updating a Data Record (LOCK$ and UPDAT$)

Every time a customer makes a payment or an order is shipped, the BALANCE field in the data record must be modified. There are two ways to handle this using MIDAS. First, a call to FIND$ can be made and the user asked to verify that the correct data record has been accessed, then LOCK$ is called if the correct record was accessed. Finally, a call to UPDAT$ alters the record. Alternatively, LOCK$ could be called first, user verification requested, and then an "unlock only call" made to UPDAT$ if the wrong record was obtained. Although the staff is careful to access the correct record, mistakes sometimes occur. Because of this, the latter sequence was choosen. The subroutine is named UPDATR and as usual only the MIDAS parts are shown. Error handling is not indicated, but the user must test ARRAY (1) for NOT FOUND, etc.

```
        SUBROUTINE UPDATR(.........
$INSERT KIDINS
        INTEGER FLAGS
        LOGICAL VERIFY
        DOUBLE PRECISION ADJSUM
                  .
                  .
                  .
C
C   SET FLAGS TO RETURN PRIMARY KEY IN BUFFER, RETURN ARRAY
C
        FLAGS=FL$KEY+FL$RET
C
C       INDEX IS 0, 'FILE#' IS 0, 0 DATA LENGTH SAYS WHOLE RECORD
C       LOCK$ DOES NOT USE KEY LENGTH
C
        CALL LOCK$(1,DBUFFR,PKEY,ARRAY,FLAGS,$9000,0,0,0)
```

```
C
C       VERIFY IS A LOGICAL FUNCTION WHICH DISPLAYS THE DATA RECORD
C       AND REQUESTS A YES-NO ANSWER.
C
        IF(.VERIFY(.....).) GO TO 2000
C
C       THIS RECORD IS NOT TO BE UPDATED, UNLOCK IT
C
        FLAGS=FL$ULK+FL$USE             /*USE ARRAY, UNLOCK ONLY
C
C       UPDAT$ DOES NOT USE KEY LENGTH
C
        CALL UPDAT$(1,DBUFFR,PKEY,ARRAY,FLAGS,$9100,0,0,0)
        RETURN
C
C       UPDATE THIS RECORD
C
C       SET FLAGS FOR PRIMARY KEY IN BUFFER, USE ARRAY
C
2000    FLAGS=FL$USE+FL$KEY
C
C       THE USER UPDATES THE VALUE OF BALANCE RETURNED BY LOCK$
C       TO INSURE THAT THE MOST RECENT COPY IS UPDATED.
C       THE AMOUNT TO BE ADDED TO BALANC IS IN ADJSUM.
C
        BALANC=BALANC+ADJSUM
C
C       THE PARAMETERS TO THIS CALL TO UPDAT$ ARE AS ABOVE
C
        CALL UPDAT$(1,DBUFFR,PKEY,ARRAY,FLAGS,$9200,0,0,0)
        RETURN
             .
             .
             .
```

In this example, the user would probably check the error return from LOCK$ to see if the record was already locked. If this is the case, it is appropriate to recycle a few times, until the record was unlocked and then proceed with the update.

Updating a Data Record (NEXT$, LOCK$ and UPDAT$)

Sometimes the staff wishes to be able to process payments by accessing the data record through the customer name and address hash. For this purpose, a slightly different routine named UPDATH is used. It is assumed that the user has obtained the user hash into SKEY1. The principle in this case is that NEXT$ is called until the correct record is found, then the file is modified using LOCK$ and UPDAT$.

```
      SUBROUTINE UPDATH(....,ALTRTN,.....
C
      LOGICAL VERIFY,AYENAY
      INTEGER FLAGS,ALTRTN
      DOUBLE PRECISION PAY
$INSERT KIDINS
C
          .
          .
          .
C
C   SET FLAGS TO USE AND RETURN ARRAY, RETURN PRIMARY
C   KEY IN BUFFER AND STOP SEARCHING ONLY WHEN
C   TERMINATED BY USER.
C
      FLAGS=FL$RET+FL$USE+FL$PLW+FL$KEY
      ARRAY(1)=-1                    /*FLAG NEXT$ TO IGNORE ARRAY
C
C   DATA WILL BE READ INTO DBUFFR
C   SEARCH WILL BE DONE ON SKEY1
C   USING INDEX 1
C   FILE# IS Ø, AS USUAL - FIRST Ø
C   PLNGTH IS Ø - RETURN FULL RECORD - 2ND Ø
C   KEYLNT IS Ø - USE FULL KEY - 3RD Ø
C
100   CALL NEXT$(1,DBUFFR,SKEY1,ARRAY,FLAGS,$9000,1,Ø,Ø,Ø)
      IF(.VERIFY(......).) GO TO 1000
C
C   AYENAY IS A FUNCTION REQUESTING A YES/NO RESPONSE FOR THE
C   MESSAGE INDICATED.
C
      IF(.AYENAY('NEXT? ',6).) GO TO 100
      GO TO ALTRTN
C
C   CALL LOCK$ USING SAME PARAMS
C
1000  CALL LOCK$(1,DBUFFR,SKEY1,ARRAY,FLAGS,$9100,1,Ø,Ø,Ø)
      BALANC=BALANC+PAY
C
C   CALL UPDAT$ USING SAME PARAMS TOO
C
      CALL UPDAT$(1,DBUFFR,SKEY1,ARRAY,FLAGS,$9100,1,Ø,Ø,Ø)
      RETURN
```

In this example, the user would probably check the error return from LOCK$ to see if the record was already locked. If this is the case, it is appropriate to recycle a few times until the record is unlocked and them proceed with the update.

Adding Records to a File (ADD1$)

A new customer is added to the file by adding the customer ID to the primary index, then adding the name and address hash using the location of the record returned by the first call to ADD1$. If an error is returned by ADD1$, if a message is printed at the users terminal, the subroutine ADDR checks to see if the record is already in file.

```
      SUBROUTINE ADDR(....,ALTRTN,....
C
$INSERT KIDINS
      INTEGER FLAGS,ALTRTN
C
          .
          .
          .
C
C     THE DATA IS IN DBUFFR AND THE KEY ESTABLISHED IN PKEY.
C     THE NAME AND ADDRESS HASH IS IN SKEY1.
C
      FLAGS=FL$RET+FL$KEY        /*KEY IN BUFFER, RETURN ARRAY
C
C     ADD RECORD TO PRIMARY INDEX
C
      CALL ADD1$(1,DBUFFR,PKEY,ARRAY,FLAGS,$8000,0,0,0,0)
      FLAGS=FL$USE                    /*USE ARRAY RETURNED.
C
C     ADD TO SECONDARY INDEX 1
C
      CALL ADD1$(1,DBUFFR,SKEY1,ARRAY,FLAGS,$8100,1,0,0,0)
      RETURN
C
8000  IF(ARRAY(1).NE.12) GO TO 8100
      WRITE 8001
8001  FORMAT('RECORD ALREADY IN FILE')
      GO TO ALTRTN
C
8100  CALL ERROUT                     /*ALL OTHER ERRORS FATAL
      END
```

Adding a Secondary Index (ADD1$)

When a backorder is to be added to the file, the customer ID must always be known. The routine to add to the backorder index is called BACK.

```
      SUBROUTINE BACK(.......,ALTRTN,...
C
$INSERT KIDINS
      INTEGER FLAGS,ALTRTN
C
        .
        .
        .
C
C   THE USER HAS PLACED THE PARTNUMBER/DATE COMBINATION
C   USED FOR THE KEY IN SKEY2.  THE CUSTOMER ID HAS BEEN
C   PLACED IN THE FIRST SIX WORDS OF DBUFFR - EQUIVALENCED
C   TO PKEY IN KIDINS.  THE THREE USER DATA WORDS HAVE BEEN
C   LOCATED IN DBUFFR(7), DBUFFR(8) AND DBUFFR(9).
C
C   SET FLAGS TO Ø, NO SPECIAL FEATURES
C   MIDAS WILL USE THE PRIMARY KEY TO LOCATE THE CORRECT
C   DATA RECORD.
C
      FLAGS=Ø
      CALL ADD1$(1,DBUFFR,SKEY2,ARRAY,0,$9000,2,Ø,Ø,Ø)
C
         .
         .
         .
```

Deleting a Data Record (DELET$)

Customer records that have been inactive for over four years are deleted from the file, after the pertinent information has been written out to magnetic tape. Deleting the data record causes all secondary index entries associated with that record to be deleted also. In this case, the file is traversed by NEXT$, a date field checked and appropriate records deleted.

```
C      USER PROGRAM TO DELETE OLD RECORDS
C
$INSERT KIDINS
      INTEGER FLAG1,FLAG2,DATE,TODATE
      DATA FLAG1/FL$RET+FL$USE+FL$KEY/
      DATA FLAG2/FL$USE/
```

```
C
C   THE DATE FIELD IN THE DATA RECORD IS CALLED 'DATE'.  THE
C   CUTOFF DATE FOR DETERMINING 'OLDNESS' IS IN 'TODATE'.
C
                .
                .
                .
       ARRAY(1)=-1                          /*TELL NEXT$ TO IGNORE ARRAY
100    CALL NEXT$(1,DBUFFR,PKEY,ARRAY,FLAG1,$9000,0,0,0,0)
       IF(DATE.GT.TODATE) GO TO 100
       CALL DELET$(1,DBUFFR,PKEY,ARRAY,FLAG2,$9100,0,0,0,0)
       GO TO 100
C
          .
          .
          .
```

Filling a Backorder Using NEXT$ and DELET$

Backorders are filled by locating the first entries in the file for the backordered part and applying the available quantity to successive backorder entries using the subroutine fill. Filled backorder entries are then deleted. In the routine, INUM specifies the quantity of an item available.

```
       SUBROUTINE FILL(.......,ALTRTN,....
$INSERT KIDINS
       INTEGER FLAG1,FLAG2,ALTRTN,INUM
       DATA FLAG1/FL$USE+FL$UKY+FL$RET+FL$KEY/
       DATA FLAG2/FL$RET+FL$UKY+FL$KEY/
C
                .
                .
                .
C
C      THIS WILL INVOLVE A PARTIAL KEY SEARCH ON THE
C      PART NUMBER HALF OF THE KEY FOR SECONDARY INDEX
C      2.  THE KEY IS IN SKEY2.  RECORDS WILL BE
C      RETURNED IN ORDER OF DATE OF BACKORDER AS THE DATE
C      IS PART OF THE KEY.  WHEN NO MORE ENTRIES WITH
C      THE CORRECT PART NUMBER EXIST NEXT$ WILL STOP
C      AS FL$PLW IS NOT SET.
C      SINCE FL$BIT IS NOT SET, THE PARTIAL KEY SEARCH
C      WILL USE THE FIRST FULL WORD OF THE KEY.
C      AS FL$UKY IS SET, THE SECOND WORD OF SKEY2 WILL BE
C      UPDATED BY MIDAS ALLOWING THE USER TO EXAMINE THE
C      DATE OF THE BACKORDER AND PROVIDING A FULL COPY
C      OF THE SECONDARY KEY FOR DELET$.
```

```
C
C       SEE IF THERE ARE ANY
C
        CALL FIND$(1,SBUFFR,SKEY2,ARRAY,FLAG2,$9000,2,0,0,1)
C
C       SBUFFR CONTAINS THE PRIMARY KEY, PLUS THE USER DATA.
C       SEE KIDINS - ABOVE
C
        GO TO 200
C
C       FOR SUBSEQUENT ENTRIES USE NEXT$
C
100     CALL NEXT$(1,SBUFFR,SKEY2,ARRAY,FLAG1,$9200,2,0,0,1)
200     IF(INUM.LT.PARTS) GO TO 3000    /*NOT ENOUGH
        INUM=INUM-PARTS
C
              .
              .
              .
C       SOME USER STUFF UNDOUBTEDLY GOES IN HERE
              .
              .
              .
C       DELETE BACKORDER ENTRY - USING DELET$
C       ON INDEX 2
C
        CALL DELET$(1,SBUFFR,SKEY2,ARRAY,FL$USE,$9100,2,0,0,0)
        GO TO 100                       /*GET NEXT
         .
         .
         .
```

Deleting a Single Secondary Index Entry

From time to time, a customer cancels a backorder. When this happens, the backorder entries for that particular customer must be deleted. Since the values of the keys in the backorder index are not guaranteed to be unique, the correct entry must be found by locating the one(s) that correspond to the correct customer. For example:

```
       SUBROUTINE UNORDR(.............
$INSERT KIDINS
       INTEGER FLAGS
          .
          .
          .
```

```
C
C  SET FLAGS TO USE AND RETURN ARRAY, KEY IN BUFFER, RETURN
C  2NDARY DATA IN SBUFFR, ON SECONDARY INDEX 2.  ALL ENTRIES
C  FOR THE CORRECT PART/DATE MUST BE EXAMINED (A FULL KEY SEARCH).
C  PRESUME THAT THE USER HAS SUPPLIED THE CUSTOMER ID (IN PKEY)
C  AND THE INFORMATION REQUIRED TO GENERATE THE SECONDARY
C  KEY VALUE, WHICH IS STORED IN SKEY2.
C
       FLAGS=FL$USE+FL$RET+FL$KEY+FL$SEC
       ARRAY=-1                          /*TELL NEXT$ TO IGNORE FLAGS
100    CALL NEXT$(1,SBUFFR,SKEY2,ARRAY,FLAGS,$9000,2,0,0,0)
C
C      COMPARE KEY VALUE IN SBUFFR (BPKEY) WITH KEY IN PKEY.  IF
C      SAME DELETE ENTRY
C
       DO 150 I=1,6
       IF(BPKEY(I).NE.PKEY(I)) GO TO 100
150    CONTINUE
         .
         .
         .
C      USER MAY WANT TO REPORT TO CALLER HERE
         .
         .
         .
       CALL DELET$(1,SBUFFR,SKEY2,ARRAY,FL$USE,$9100,2,0,0,0)
       GO TO 100
           .
           .
           .
```

AN OFF-LINE PROGRAM

Here is an example of a general build/add program using the MIDAS subroutines PRIBLD and BILD$R. The data for the program is of no interest as it is generated by the program. The important part is the calls to MIDAS. In this instance, the data will be generated so that it is sorted on primary key order. If the build path is chosen, this fact can be used to create the file more quickly, using PRIBLD for the primary index and BILD$R for the secondary index subfiles. (There are two secondary index subfiles to be built.) If the add path is chosen, BILD$R must be used to build the primary as well as the secondary index subfiles, since PRIBLD would destroy the existing file.

```
C  OFFLINE TEST PROGRAM EXERCISING PRIBLD AND BILD$R
C
C   THE COMMAND FILE TO BUILD THE PROGRAM IS C_NDAV.
C   THE COMMAND FILE TO BUILD THE TEMPLATE IS C_DAVB
C
C   THIS PROGRAM BUILDS A MIDAS FILE WITH TWO SECONDARY INDICES.
C
C   PRIMARY INDEX - KEY IS ASCII, LENGTH 6 CHARACTERS.
C
C   DATA RECORD IS VARIABLE LENGTH - CONTROLED IN THIS PROGRAM BY
C          THE VARIABLE 'LGH'.
C
C   FIRST SECONDARY INDEX - KEY IS ASCII, LENGTH 6 CHARACTERS.
C          THERE IS NO SECONDARY DATA.
C
C   SECOND SECONDARY INDEX - KEY IS ASCII, LENGTH 4 CHARACTERS.
C           THERE IS NO SECONDARY DATA.
C
C   THIS PROGRAM ALLOWS THE USER TO BUILD A FILE FROM 'SCRATCH',
C   ADD TO AN EXISTING FILE OR DO BOTH.  THE USER INTERACTIVELY
C   SELECTS WHICH IS TO BE DONE.  THE USER IS ALSO ASKED TO SUPPLY
C   THE NAME OF A LOG/ERROR
C   FILE TO WHICH MILESTONE AND ERROR MESSAGES WILL BE WRITTEN.
C
C ****************************************************************
C
C    ARRAYS RELATING TO THE DATA AND KEYS
C
       DIMENSION K3(3),K1(3),K2(2),IDATA(80),IBUFF(89)
C
C      THE COMMUNICATIONS AREAS FOR PRIBLD AND BILD$R
C
       INTEGER SEQFLG,JTEMPS(2),KTEMPS(2),LTEMPS(2),JARRAY(14)
       DIMENSION K14(2),K23(1)
C
C      EQUIVALENCE THINGS IN THE DATA BUFFER
C
       EQUIVALENCE (IBUFF(1),K3(1)),(IBUFF(4),K1(1))
       EQUIVALENCE (IBUFF(7),K2(1)),(IBUFF(10),IDATA(1))
       EQUIVALENCE (IBUFF(9),LGH)
       EQUIVALENCE (K14,K1(2)),(K23,K2(2))
C
C    YOU NEED A COPY OF OFFCOM.  OURS IS IN THE LIBRARY UFD
C
$INSERT LIB>OFFCOM
C
C      SET UP A 'DATA RECORD'
C
       DATA IDATA/80*'AB'/
C
C      SET UP OF UNIT NUMBERS.  THE VARIABLES ISUNIT, ISSAM,
C      ISDAT AND ISDAM ARE DEFINED IN OFFCOM.
```

```
C
      ISUNIT=1
      ISSAM=2
      ISDAM=3
      ISDAT=4
C
C   SET THE COMMUNICATIONS FLAG TO INITIATE THE BUILD ROUTINES
C
      SEQFLG=0              /*PRIBLD NEEDS ONLY ONE WORD
      KTEMPS(1)=0           /*BILD$R NEEDS TWO WORDS
      LTEMPS(1)=0
      JTEMPS(1)=0           /*IN CASE USER IS ONLY GOING TO ADD TO FILE
C
C     CLOSE ALL UNITS BUT 6, THEN OPEN THE MIDAS TEMPLATE
C
      CALL SYSINI
      CALL SEARCH(:4001,'DATBAS',ISUNIT)
      CALL FILSET           /*SET UP THE DESCRIPTOR SUB FILE
C
C
C     INIT KEYS AND DATA RECORD LENGTH
C
      K1(1)='AA'
      K1(2)='AA'
      K2(1)='AA'
      LGH=9
      J1=0                          /*A FLAG FOR ADD ONLY PATH
      CALL TNOUA('HOW MANY: ',10)
      CALL TIDEC(NUM)
C
C     IF USER ANSWERED 0, ADD TO EXISTING FILE
C
      RNUM=NUM
      CALL ERROPN(5)                    /*GET AN ERROR FILE
      IF(NUM.EQ.0) GO TO 5000
      CALL KX$TIM(000000,5,0,0)
C
C     PRIBLD FLAGS BILD$R TO USE A DATA RECORD POSITION SET UP
C     BY PRIBLD SO IN THIS CASE JARRAY IS A FAKE.
C
      JARRAY(1)=-1
C
C     MAIN LOOP TO BUILD THE FILE
C
      DO 500 J1=1,NUM
C
C     SET UP THE DATA AND KEYS
```

```
C
      IK1=RT((K3(3)+IK1),10)
      IK2=(IK1/25)+1
      RK3=RK3+1
      IRK3=RK3
      ENCODE(5,100,K3) RK3
100   FORMAT(B'Z#####')
      ENCODE(3,110,K14) IK1
110   FORMAT(B'####')
      K1(2)=RT(K1(2),8)+LS(:301,8)
      ENCODE(2,120,K23) IK2
120   FORMAT(B'##')
C
C     SET THE LENGTH OF THE DATA RECORD
C
      LGH=LGH+1
      IF(LGH.GT.89) LGH=1+9
C
C     ADD THE PRIMARY INDEX AND DATA ENTRY
C
      CALL PRIBLD(SEQFLG,K3,IBUFF,LGH,$999,RNUM)
C
C     WRITE THE SECONDARY KEYS
C
      CALL BILD$R(KTEMPS,K1,K3,0,JARRAY,1,$450,RNUM)
      CALL BILD$R(LTEMPS,K2,K3,0,JARRAY,2,$450,RNUM)
C
C     EVERY 1024 RECORDS CALL THE MILESTONE ROUTINE
C
      IF(AND(J1,:1777).EQ.0) CALL KX$TIM(INTL(I),5,0,0)
500   CONTINUE
C
C     SET THE VALUES OF THE COMMUNICATIONS FLAGS TO 2 TO
C     TELL THE BUILD ROUTINES TO FINISH UP THE INDICES
C
      SEQFLG=2
      KTEMPS(1)=2
      LTEMPS(1)=2
C
C     MAKE THE FINAL CALLS
C
      CALL PRIBLD(SEQFLG,K3,IBUFF,LGH,$999,RNUM)
      CALL BILD$R(KTEMPS,K1,K3,0,JARRAY,1,$450)
      CALL BILD$R(LTEMPS,K2,K3,0,JARRAY,2,$450)
C
C     MAKE A CALL TO THE MILESTONE ROUTINE
C
      CALL KX$TIM(INTL(I),5,'END PASS 1',5)
C
C     PROCEED TO THE 'SECOND PASS'
C
      JTEMPS(1)=0
      LTEMPS(1)=0
      KTEMPS(1)=0
      GO TO 6000
```

```
C
5000    CALL TNOUA('STARTING NUMBER: ',17)
        CALL TIDEC(IRK3)
        RK3=IRK3
6000    CALL TNOUA('HOW MANY: ',10)
        CALL TIDEC(NUM)
        IF(NUM.EQ.0) GO TO 7000
        CALL KX$TIM(INTL(J1),5,'PASS 2',3)
C
C       MAIN LOOP TO ADD TO THE FILE
C
        DO 6500 J1=1,NUM
C
C       SET UP THE DATA RECORD
C
        IK1=RT((K3(3)+IK1),10)
        IK2=(IK1/25)+1
        RK3=RK3+1
        IRK3=RK3
        ENCODE(5,100,K3) RK3
        ENCODE(3,110,K14) IK1
        K1(2)=RT(K1(2),8)+LS(:301,8)
        ENCODE(2,120,K23) IK2
C
C       SET UP DATA RECORD LENGTH
C
        LGH=LGH+1
        IF(LGH.GT.89) LGH=1+9
C
C       ADD DATA AND SECONDARIES TO THE FILE.  CAN'T USE PRIBLD TO ADD
C       TO A FILE.
C
        CALL BILD$R(JTEMPS,K3,IBUFF,LGH,JARRAY,0,$450)
        CALL BILD$R(KTEMPS,K1,K3,0,JARRAY,1,$450)
        CALL BILD$R(LTEMPS,K2,K3,0,JARRAY,2,$450)
C
C       EVERY 1024 RECORDS CALL THE MILESTONE ROUTINE
C
        IF(AND(J1,:1777).EQ.0) CALL KX$TIM(INTL(J1),5,0,0)
6500    CONTINUE
C
C       SET THE COMMUNICATIONS FLAGS TO 2 AND MAKE FINAL CALLS
C
        JTEMPS(1)=2
        CALL BILD$R(JTEMPS,K3,IBUFF,LGH,JARRAY,0,$450)
        KTEMPS(1)=2
        CALL BILD$R(KTEMPS,K1,K2,0,0,1,$450)
        LTEMPS(1)=2
        CALL BILD$R(LTEMPS,K2,K3,0,0,2,$450)
```

```
C
C       MAKE A FINAL CALL TO THE MILLESTONE ROUTINE
C
7000    CALL KX$TIM(INTL(J1),5,'END',2)
C
C       CLOSE ALL FILES EXCEPT UNIT 6
C
        CALL SYSINI
        CALL EXIT
C
C       GO AN ERROR ON A CALL TO BILD$R.  PRINT FILE HANDLER ERROR
C       CODE - IF ANY - AND ABORT.
C
450     CALL FILHER(JARRAY(1),0)
C
C       GOT AN ERROR, CALL FILERR TO PRINT MESSAGE AND ABORT
C
999     CALL FILERR('BDAVIS','BAD PRIBLD',10,0)
        END
```

APPENDIX A

CONDITION CODES

NONFATAL CODES

The following codes are non-fatal and indicate that the user program may continue as long as the program correctly interprets the code.

Code	Meaning
0	No error
1	There may be a synonym index entry for the user-supplied key. The user may not have located the exact entry desired and the index should perhaps be searched further to check.
7	Entry not found. This error condition terminates any sequence of calls to NEXT$. The contents of the communication array must be considered to be invalid in this case.
10	User is attempting to lock a data entry for update, and the entry has already been locked by another user. The current user must wait until the entry becomes available. It is possible that an entry may have remained locked because of a system crash or program failure. If it is suspected that this may be the case, contact an analyst for help in unlocking such records.
11	User has not locked a data record before attempting to update it. The user must lock the record with a call to LOCK$, then a call to UPDAT$ must be made using the copy of the communications array returned by LOCK$. Intervening calls to other MIDAS routines using this copy of the communications array may interfere with the LOCK/UPDATE process.
12	User is attempting to add a data entry whose primary key already exists in the data file or is trying to add a duplicate secondary index entry to an index that does not permit duplicate keys. If appropriate, the user should delete the existing entry before attempting to add the new one.

DISK ERROR CONDITION CODES

The following error condition codes indicate failure on an attempt to access a disk file. They are generally fatal errors.

Note

It may be appropriate to treat error codes 22 and 24 as nonfatal errors if the user's system is configured for more readers than writers to disk files. In this case, these codes may mean that another user has the required file open and will release it shortly. A call to the system routine RECYCL may return with the file available for a new user.

The user must determine whether the disk error reported on his terminal is due to a disk failure or to a software problem. If the user cannot locate the source of a software problem in the program, he should contact his field analyst for help in determining other possible sources of the problem. Disk error codes are as follows:

Code	Meaning
20	Error on attempt to write to disk. This is fatal.
21	Error on attempt to read from disk. This is fatal.
22	A needed segment file is open for input/output by another user. The current user cannot continue until the file becomes available.
23	An error was encountered while trying to open a segment file. This is a fatal error.
24	The MIDAS file is already open for input/output by another user. The current user cannot add any new entries to the file until the other user has released the file.
25	An error was encountered on an attempt to open the MIDAS file for input/output so that a new entry could be added. This is a fatal error.
26	An error on attempt to examine (read) the disk. This is fatal.
27	Error on attempt to add a data entry. This is fatal.
28	Error on attempt to close a file. This is fatal.

FILE HANDLER CONDITION CODES

The following error codes are generated as a result of incorrectly supplied arguments when calling one of the MIDAS file handler routines. The user must correct his program and try again.

Code	Meaning
30	The user has called either NEXT$ or LOCK$ without specifying that the communication array is to be returned. This can be corrected by setting the flag FL$RET in the MIDAS flag parameter for these calls.
31	The user has not supplied a copy of the communications array on a call to UPDAT$. This can be corrected by setting the flag FL$USE in the MIDAS flag word and supplying a copy of the array generated by a call to LOCK$.
32	The user has supplied a data length as the parameter plenth that is inconsistent with the length maintained in the MIDAS file. This generally occurs if plenth is too long.
33	The user has supplied a bad copy of the communications array for use by the MIDAS file handler for subsequent processing. This occurs if the user attempts to use a copy of the array which generated an error on a previous call to the file handler. There are also other possible causes.
34	The user has attempted to call NEXT$ through direct access.
35	The user has attempted to add a data entry to a direct access file through the primary index or has tried to add a data entry through direct access to a file not set up for direct access.

MIDAS ERROR CONDITION CODES

The user may encounter error codes 42, 43, 44, 45, 46 or 48. These codes reflect problems within MIDAS itself or in the MIDAS file. The user must note the value of the code carefully and report it to his field analyst.

FILE SIZE CONDITION CODES

The following error codes indicate that the MIDAS file is not configured to be large enough for any additional records. The user should contact his field analyst for a possible solution.

Code	Meaning
51	The addition of a data record will make the data subfile too large.
52	The addition of another index entry will make the index subfile too large.

APPENDIX B

CREATK MAXIMUM OPTIONS DIALOG

This appendix explains the MAXIMUM OPTIONS version of the CREATK dialog. The format is similar to the example in Section 2. For an explanation of the symbols and other pertinent information, refer to the CREATK Utility section. (Section 2)

Note

The opening statements of the CREATK dialog are not shown. This section commences with line 38.

(Line 38)	FILE NAME?	filename or treename	(Goes to Line 39)
		(other)	(Repeats Line 38)
(Line 39)	NEW FILE?	YES	(Goes to Line 40)
		NO	(Goes to Line 23)
		(other)	(Repeats Line 39)

If the response to line 39 is NO, the MINIMUM OPTIONS dialog will be invoked. Refer to Section 2 for subsequent steps.

(Line 40)	DIRECT ACCESS?	YES	(Goes to Line 69)
		(other)	(Repeats Line 40)

If CREATK was not built (i.e., copied from the master disk) with direct access support, then line 40 will not be included in the dialog.

(Line 41)	KEY:	B	(Goes to Line 43)
		A	(Goes to Line 43)
		T	(Goes to Line 44)
		L	(Goes to Line 44)
		D	(Goes to Line 44)
		S	(Goes to Line 44)
		(other)	(Repeats Line 42)

For an explanation of the key types, refer to the MINIMUM OPTIONS CREATK dialog in Section 2.

(Line 43)	KEY SIZE=:	B (number of bits or bytes)	(Goes to Line 44)
		W (number of words	(Goes to Line 44)
		(other)	(Repeats Line 43)
(Line 44)	DATA SIZE=:	(number of words)	(Goes to Line 45)
		(CR)	(Goes to Line 45)
		(other)	(Repeats Line 44)

Note

For Line 44, if the number of words is specified; that number of record must not include the key size; i.e., it must specify the size of the data only. If the user responds to Line 44 with a carriage return (CR), then it indicates that the file has variable length records.

(Line 45)	DOUBLE LENGTH INDEX?	YES	(Goes to Line 46)
		NO	(Goes to Line 46)
		(other)	(Repeats Line 45)
(Line 46)	PRIMARY INDEX FIRST LEVEL OF INDEX		(Goes to Line 47)
(Line 47)	BLOCK SIZE=:	(number of words in block)	(Goes to Line 48)
		(CR)	(Goes to Line 48)
		(other)	(Repeats Line 47)

Note

If the KEY TYPE is not A or B, and the user specifies a number of words in the block, then the dialog returns to Line 40.

(Line 48)	KEY SIZE=:	B(number of bits or bytes)	(Goes to Line 49)
		W(number of words)	(Goes to Line 49)
		(other)	(Repeats Line 48)
(Line 49)	SECOND LEVEL OF INDEX		(Goes to Line 50)
(Line 50)	BLOCK SIZE=:	(number of bits)	(Goes to Line 51)
		(CR)	(Goes to Line 51)
		(other)	(Repeats Line 50)
(Line 51)	KEY SIZE=:	(number of bits or bytes)	(Goes to Line 52)
		(number of words)	(Goes to Line 52)
		(other)	(Repeats Line 51)

(Line 52)	LAST LEVEL OF INDEX		(Goes to Line 53)
(Line 53)	BLOCK SIZE=:	(number of words in block)	(Goes to Line 54)
		(CR)	(Goes to Line 54)
		(other)	(Repeats Line 53)
(Line 54)	SECONDARY INDEX		(Goes to Line 55)
(Line 55)	INDEX NO.?	(numeric 1-19)	(Goes to Line 54)
		(CR)	(Goes to Line 54)
		(other)	(Repeats Line 55)

Note

Double length indexes are optional, according to the version of CREATK. If they are not supported, CREATK continues with line 57. Refer to Section 2.

(Line 56)	DOUBLE LENGTH INDEX?	YES	(Goes to Line 57)
		NO	(Goes to Line 57)
		(other)	(Repeats Line 56)
(Line 57)	DUPLICATE KEYS PERMITTED?	YES	(Goes to Line 58)
		NO	(Goes to Line 58)
		(other)	(Repeats Line 57)
(Line 58)	KEY TYPE:	B	(Goes to Line 59)
		A	(Goes to Line 59)
		I	(Goes to Line 59)
		L	(Goes to Line 59)
		D	(Goes to Line 59)
		S	(Goes to Line 59)
		(other)	(Repeats Line 58)
(Line 59)	FIRST LEVEL OF INDEX		(Goes to Line 60)
(Line 60)	BLOCK SIZE=:	(number of words)	(Goes to Line 61)
		(CR)	(Goes to Line 61)
		(other)	(Repeats Line 60)

Note

If KEY TYPE is not equal to A or B, and the number of words is specified in response to Line 60, then CREATK goes to Line 62.

(Line 61)	KEY SIZE=:	B(number of bits in Key)	(Goes to Line 62)
		W(number of words in Key)	(Goes to Line 62)
		(other)	(Repeats Line 61)
(Line 62)	SECOND LEVEL OF INDEX		(Goes to Line 63)

Note

If KEY TYPE is not equal to A or B, and the number of words is specified in response to Line 60, then CREATK goes to 65.

(Line 63)	BLOCK SIZE=:	(number of words)	(Goes to Line 64)
		(CR)	(Goes to Line 64)
		(other)	(Repeats Line 63)
(Line 64)	KEY SIZE=:	B(number of bits in key)	(Goes to Line 65)
		W(number of words in key)	(Goes to Line 65)
		(other)	(Repeats Line 64)
(Line 65)	LAST LEVEL OF INDEX		(Goes to Line 66)
(Line 66)	BLOCK SIZE=:	(number of words in block)	(Goes to Line 67)
		(CR)	(Goes to Line 67)
		(other)	(Repeats Line 66)

Note

If KEY TYPE is not A or B, and the number of words is specified in response to Line 66 CREATK goes to Line 68.

(Line 67)	KEY SIZE=:	B(number of bits)	(Goes to Line 68)
		W(number of words)	(Goes to Line 68)
		(other)	(Repeats Line 67)
(Line 68)	USER DATA SIZE=:	(number of words)	(Goes to Line 69)
		(CR)	(Goes to Line 69)
		(other)	(Repeats Line 68)

Note

If the user has responded to YES to the question DIRECT ACCESS in Line 40, the following questions are asked; otherwise

(Line 69)	DATA SUBFILE QUESTIONS	(Goes to Line 70)

(Line 70)	KEY TYPE:	B	(Goes to Line 71)
		A	(Goes to Line 71)
		I	(Goes to Line 72)
		L	(Goes to Line 72)
		D	(Goes to Line 72)
		S	(Goes to Line 72)
		(other)	(Repeats Line 71)
(Line 71)	KEY SIZE=:	B(number of bits)	(Goes to Line 72)
		W(number of words)	(Goes to Line 72)
		(other)	(Repeats Line 71)
(Line 72)	DATA SIZE=:	number of words in entry	(Goes to Line 73)

Note

The user must include the numeric value of the key size in above.

(Line 73)	NUMBER OF ENTRIES TO ALLOCATE?		
		(number)	(Goes to Line 41)
		(other)	(Repeats Line 73)

INDEX